UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València


# Videojuego Beast's Retreat en Unity con C# integrando RT-DESK: Gameplay

Proyecto Final de Carrera

Ingeniería Informática

**Autor**: Javier Baixauli Herraez

**Director**: Ramón Pascual Mollá Vayá

2013/2014

A Santiago Martínez Gómez,

porque sin nuestro mutuo apoyo

ninguno habría llegado hasta el final.

# Resumen

Este PFC consiste en el desarrollo de un videojuego mediante C# a través del motor gráfico Unity, que permite realizar desarrollos en diferentes plataformas, e integrar el kernel para generación de eventos discretos RT-DESK.

Bajo el nombre de Beast's Retreat, a modo de resumen podemos definir este juego como un *tower defense* en el cual los enemigos se irán sucediendo en oleadas con el fin de destruir una base que los usuarios deben defender. Para defender esta base el usuario deberá utilizar una serie de torres con las cuales será capaz de lanzar proyectiles en la dirección, ángulo y distancia deseada.

El objetivo del PFC, una vez acabado el juego, es modificar ligeramente el código mediante la integración de funciones con el kernel RT-DESK. A través de estas funciones podremos mejorar el rendimiento del videojuego mediante las herramientas que nos proporciona. El RT-DESK se encuentra actualmente codificado en C++, por lo que para poder utilizarlo en Unity habrá que hacer una adaptación al lenguaje C#.

El gameplay del juego constará de varios niveles con elementos comunes. Se basa en superar distintas hordas de enemigos utilizando las torres elegidas anteriormente para acabar con todos ellos. Se deberá implementar todo el comportamiento tanto de las torres, enemigos, proyectiles, efectos especiales, interacción con los escenarios e interacción con la interfaz básica del gameplay, es decir, botones para el control de las torres, animaciones de cámara y movimientos de los personajes.

**Palabras clave:** videojuego, Unity, C#, RT-DESK, 2.5D, tower defense.

# Tabla de contenidos

# 1. Introducción

## 1.1. Propósito

Se desea desarrollar un videojuego que permita a los jugadores vivir una experiencia "*Tower defense*" teniendo total control sobre la trayectoria de los disparos de las torres así como pudiendo personalizar las torres a utilizar a partir de uniones entre diferentes elementos.

Este juego servirá como prueba para la integración de RT-DESK , un simulador de eventos discretos orientado a simular sistemas mediante teoría de colas, en el motor de videojuegos multiplataforma Unity.

## 1.2. Motivación

La principal motivación detrás de este proyecto es obtener un producto tangible con el cual poder acceder a la industria del videojuego. Los puestos para este sector están muy demandados por lo que es necesario emprender proyectos antes de entrar a un puesto de trabajo.

El motor para desarrollo de videojuegos Unity está cobrando importancia en el sector independiente gracias a su capacidad multiplataforma que permite desarrollar videojuegos tanto para Android e iOS como para web, sistemas Windows e incluso PS3. Si sumamos a esta característica el hecho de su código abierto y la cantidad de documentación que se puede encontrar, este motor se convierte en una buena opción para empezar a desarrollar videojuegos.

Para añadir un elemento diferenciador de cualquier otro videojuego en Unity se decidió hacer una traducción de RT-Desk para poder integrar este simulador de eventos discretos dentro del juego y así optimizar la carga computacional del juego rompiendo el esquema típico que suelen utilizar todos los motores gráficos.

Algunas de las mejoras por utilizar RT-Desk podrían ser marcar la diferencia entre el consumo de batería en un dispositivo móvil o lograr una potencia de cálculo mucho mayor pudiendo así manejar más elementos a la vez.

El tipo de videojuego a desarrollar para integrar este simulador de eventos está en parte motivado por la necesidad de ver la mejora de potencia en las trayectorias de los proyectiles de las torres. La otra motivación detrás del juego es la capacidad de mantenerte enganchado que tienen este tipo de juegos basados en aguantar hordas de enemigos y el factor adictivo de conseguir todas las combinaciones de torres. De esta forma el juego tendría una fácil

monetización basándonos en la compra de los diferentes elementos que forman las torres.

## 1.3. Influencias

Las influencias de las que bebe el juego se podrían dividir en 3:

Por una parte tenemos las influencias de juegos tipo *Tower defense*. Estos juegos son muy populares para juegos web y dispositivos móviles. Podemos ver varios en ejemplos en juegos como *Kingdom Rush*, *Radiant Defense* o *Jelly Defense*.

**Ilustración 1: Captura del juego Kingdom Rush**

Otra de las influencias directas es el juego web *Bowmaster*, donde se podía manejar a un arquero que lanzaba flechas creando trayectorias en 2D.


**Ilustración 2: Captura del juego Bowmaster**

La última de las influencias que nos queda por nombrar está relacionada con la creación de elementos de la torre por unión de varios elementos básicos. Esto es influencia del juego *Doodle God*.


**Ilustración 3: Captura del juego Doodle God**

## 1.4. Importancia del sector en la informática

La industria del videojuego siempre ha estado íntimamente ligada a la informática. Conforme la informática ha ido avanzando, los videojuegos también lo han hecho hasta llegar a ser como hoy en día.

La industria de los videojuegos ha crecido en los últimos años notablemente, apareciendo múltiples empresas desarrolladoras independientes. Este incremento en los juegos indie se debe principalmente al auge de los dispositivos móviles, portales de juegos online y plataformas de venta digital.

Según datos aportados por la AEVI (Asociación Española de Videojuegos), en nuestro país, hoy el videojuego se posiciona como la principal opción de ocio para cada vez más segmentos poblacionales. Así, la penetración social del videojuego continúa aumentando en nuestro país, y es que a día de hoy el 62% de los menores de edad y el 24% de los adultos españoles se declaran ya usuarios habituales.

En el 2011 el videojuego generó en España un consumo superior a los 980 millones de euros (fuente Gfk), en donde se incluyen la comercialización de software, hardware y periféricos,.. De esta cifra, el 51% del consumo

corresponde a software (videojuegos), el 38% a hardware (consolas) y el 11% restante a periféricos.

Según los datos registrados por Pricewaterhouse Coopers en su informe Global Entertainment and Media Outlook: 2011–2015, en 2010 el valor del mercado mundial del videojuego ascendió a 56.000 millones de euros y crecerá hasta los 82.000 millones en 2015, en base a una tasa anual de crecimiento compuesto del 8,2 por ciento.

Se prevé además un desarrollo exponencial del mercado en los países emergentes. La consultora Ovum en su informe Digital Games Outlook 2011–16 señala que para 2016 el mercado Asia Pacífico alcanzará los 30.300 millones de dólares, superando la cifra de los principales mercados occidentales.

Las conclusiones de ambos informes están en la línea de las expectativas de la industria. El desarrollo de la banda ancha, los MMO, las nuevas plataformas y opciones de juego, una lucha eficaz contra la piratería y el comienzo de la recuperación económica darán un nuevo impulso a esta industria.

## 1.5. Visión global

En este apartado se va a dar brevemente una visión global del documento.

En primer lugar se podrá ver una **Descripción** del juego en su fase conceptual. Esta descripción se verá complementada con el **Anexo A: Game Design Document.**

A continuación se dará una descripción sobre el motor **Unity** para aclarar ciertos términos y conceptos que se utilizarán a lo largo del documento.

El desarrollo de un videojuego sigue una estructura similar a la de cualquier desarrollo software por lo cual tendrá una fase de **análisis**, otra de **diseño**, **implementación**, pruebas y mantenimiento. Por lo que los siguientes puntos relatarán los puntos clave del desarrollo.

Una vez documentado todo el proceso de desarrollo del videojuego siguiente la metodología empleada por Unity se pasará a explicar el funcionamiento de **RT-Desk** y como se ha traducido para poder ser integrado con el proyecto en Unity.

Finalmente se hará una relación entre los **conocimientos aplicados durante la carrera** al desarrollo de videojuegos, para seguir con las **conclusiones** aprendidas a partir del proyecto y la **bibliografía**.

# 2. Descripción

## 2.1. Perspectiva del producto

El juego Beast's Retreat se crea con el fin de entretener a los usuarios e incentivar sus ganas de jugar mediante el descubrimiento de nuevos elementos y combinaciones de torres.

En el juego se podrán definir las torres mediante la unión de tres elementos básicos: la estructura de la torre, el personaje y su arma. Cada uno de estos elementos se creará a partir de la unión de elementos básicos que podrán ser adquiridos en la tienda del juego por monedas que se reciben al pasarse las misiones.

Una vez se ha decido la formación de torres a utilizar se puede iniciar un nivel y utilizar las torres para disparar a las hordas de enemigos y acabar con ellos.

De esta forma, el conjunto de Gameplay y Editor forman una experiencia única con la cual se pretende conseguir que el producto sea un éxito y pueda ser ampliado mediante nuevos elementos para las torres alargando así su ciclo de vida.

## 2.2. Tipo de juego

Como ya se ha dicho en las influencias este juego es una mezcla de *Tower Defense* y de *Acción.*

**Tower defense** o **TD** es un subgénero de los videojuegos de estrategia en tiempo real. El objetivo es lograr que las unidades enemigas no lleguen a cruzar el mapa, para lograrlo se deben construir torres que las atacan al pasar. Tanto los enemigos como las torres tienen diferentes habilidades y costes. Al eliminar una unidad enemiga se reciben puntos o dinero que debe ser utilizado para construir o mejorar torres.

Un **videojuego de acción** es un videojuego en el que el jugador debe usar su velocidad, destreza y tiempo de reacción. El género de acción es el más amplio del mundo de los videojuegos. En nuestro caso sería una variante de juego de disparos o lanzamientos en 2D.

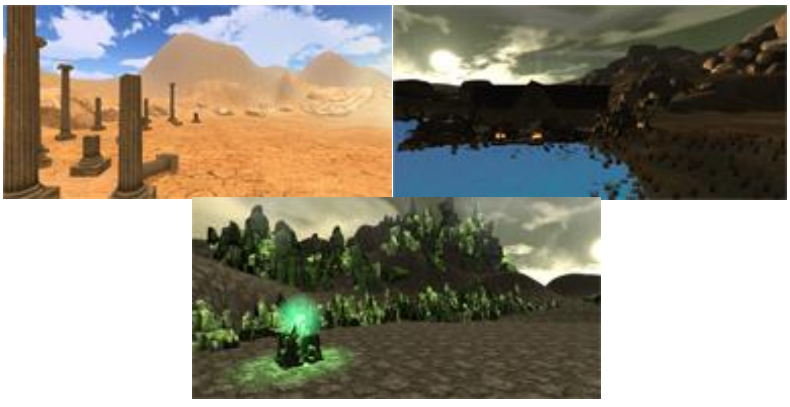
## 2.3. Funciones o características del juego

Las funciones o características del juego se pueden separar en dos bloques:

- Gameplay (Realizado por mí en este proyecto)
  - Selección de torre
  - Maquina de estados determinista para cada torre
  - Controlador de recargas
  - Física para proyectiles
  - Movimiento de cámara limitado
  - Cambio de perspectiva
  - Interfaz básica durante el juego
  - Enemigos en hordas
  - Controlador de hordas
  - Generador de partículas para crear efectos elementales
  - Mostrar daño
  - Headshots
  - Controlador de colisiones
- Editor (Realizado por Santiago Martínez Gómez en su proyecto)
  - Introducción animada
  - Navegabilidad entre escenarios
  - Persistencia de datos
  - Combinación de elementos
  - Creación de torres
  - Selección de estrategias o bases
  - Mapa con selección de escenario
  - Tienda
  - Ofertas
  - Créditos

## 2.4. Escenarios

Para este proyecto se han creado tres escenarios completamente jugables. Estos escenarios corresponden con *El desierto de Janna*, *Las montañas esmeralda* y *El palacio de Orinthalian.* Estos escenarios aparecen más detallados en el Anexo A, pero podemos ver unas imágenes a continuación.



## 2.5. Modo de juego

La forma de interactuar con el juego está basada únicamente en el ratón.

Para seleccionar una torre solo debemos pulsar en cualquiera de los botones de selección de torre que podemos ver en la imagen con la etiqueta roja . Están ordenados al igual que las torres y contienen imágenes representativas del arma a utilizar.

Para cambiar de vista y de fila de torres solo debemos pulsar en el botón de cambio como podemos ver en las imagen con la etiqueta morada.

Para accionar la siguiente horda solo debemos pulsar el botón con la etiqueta azul.

Para pausar el juego o controlar el sonido tenemos los botones con la etiqueta verde.


**Ilustración 4: Interfaz de Gameplay**

Para disparar debemos pulsar en cualquier punto del escenario que no sea un botón y la barra de carga empezará a cargar, cuando se suelte la torre disparará.

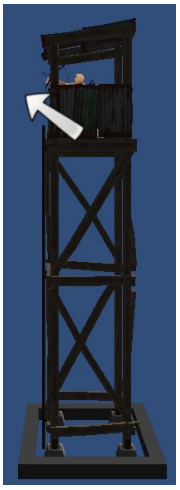


Ilustración 5: Barra de potencia descargada

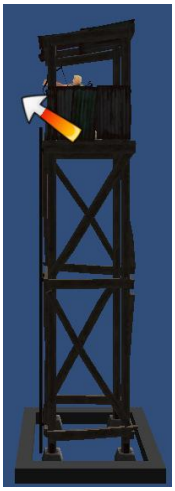


Ilustración 6: Barra de potencia cargando

Por último para manejar la cámara solo debemos pulsar con el botón derecho y arrastrar para movernos en los ejes x e y. Para hacer zoom debemos utilizar la rueda del ratón.

# 3. Unity

## 3.1. Comparativa

En la actualidad se pueden encontrar una gran variedad de motores de juego, cada uno con sus características distintivas, sus ventajas e inconvenientes. De entre todos ellos nos hemos centrado en aquellos que fueran multiplataforma.

*Unreal Engine* es un motor de juego de PC y consolas creados por la compañía Epic Games. Implementado inicialmente en el shooter en primera persona *Unreal* en 1998, siendo la base de juegos como *Unreal Tournament*, *Deus Ex*, *Turok*, *Tom Clancy's Rainbow Six: Vegas*,*America's Army*, *Red Steel*, *Gears of War*, *BioShock*, *BioShock 2*,*BioShock Infinite, Star Wars Republic Commando*, *Batman: Arkham Asylum* o *Mass Effect*. También se ha utilizado en otros géneros como el rol y juegos de perspectiva en tercera persona. Está escrito en C++, siendo compatible con varias plataformas como PC, Apple Macintosh y la mayoría de consolas. Unreal Engine también ofrece varias herramientas adicionales de gran ayuda para diseñadores y artistas.

Ilustración 7: Logo Unreal Engine

Unreal ofrece varias versiones de pago pero también tiene una de uso libre no comercial. Esta versión se llama *UDK*. Es muy buen motor gráfico para personas que quieran aprender un poco de desarrollo de videojuegos AAA o con grandes acabados. En sí, es una edición gratuita del *Unreal Engine 3* Permite exportar a Pc o Móviles. Sin embargo, es para uso no comercial. Por lo que puede ser utilizado como Herramienta de entrenamiento, o incluso compartir los proyectos en los que trabajes.

Ilustración 8: Logo UDK

Características:

- Tiene un **completo editor de niveles**, se puede incluso crear algunos elementos avanzados con el editor.
- **Motor de Renderizado de alta calidad**. Con iluminación en tiempo real, partículas y fluidos espectaculares.
- Utiliza como motor de **Scripting Kismet** que trabaja en parte con árboles de decisiones.
- Un potente **Motor de Físicas**.
- Permite crear **cinemáticas con un excelente acabado** cinematográfico.
- **Sistemas de Sonido y Animación** más completos con características avanzadas.
- **Exportación a multiplataforma** (PC, iOS).
- Lo puedes descargar directamente y completamente gratis desde su sitio oficial.

Ilustración 9: Logo CryEngine

Es el potente motor gráfico desarrollado por la empresa *Crytek*, creadores de potentes obras artísticas como *Crysis*. Posee grandes características. Es utilizado en casos muy profesionales y debido a su extensión la curva de aprendizaje se hace un poco más compleja. Es un Motor gráfico Gratuito para usos no comerciales al igual que el *UDK*. Sin embargo, si quieres distribuir tu proyecto, lo puedes hacer al repartir el 20% de las ganancias con *Crytek*. Permite exportar a plataformas como Pc, Xbox 360 y Ps3.

Características:

- Uno de los **mejores motores de Render gratuitos**. La iluminación, las texturas, partículas y fluidos son Hiperrealistas. Incluso las texturas se pueden modificar dentro del motor dándoles un aspecto de deterioro.
- Contiene un **potente editor facial** con el que se logran rostros realistas.
- **Sistema integrado de Pesos y esqueletos** para los personajes, permitiendo unas herramientas de animación más potentes.
- **Físicas Avanzadas**. Para crear mundos más realistas y colisiones perfectas.
- Posibilidad de crear videojuegos en **3D estereoscópico**.
- Posibilidad de crear un **ambiente sonoro realista** con capacidad para canales 7.1 de alta calidad.
- Lo puedes descargar directamente y completamente gratis desde su sitio oficial.

Ilustración 10: Logo Unity

*Unity* es un excelente motor gráfico por muchas razones. Es fácil de usar, permite exportar a múltiples plataformas, se actualiza constantemente con nuevos contenidos y opciones, permite desarrollar en 2D y 3D, su curva de aprendizaje es muy rápida y es completamente Gratis.

Características:

- Exportación Multiplataforma gratuita (móvil, web, PC y Mac, PS3).
- Scripting en Java Script, C# ó Boo. Contiene un editor de código.
- Soporta gran cantidad de paquetes 3D y texturas de múltiples extensiones.
- Soporte para creación de redes y juego en línea.
- Editor de Terrenos y vegetación incorporada.
- Creación de Videojuegos 2D y 3D.
- Gestor de animaciones, arboles de mezcla y máquinas de estado.

Lo puedes descargar directamente y completamente gratis desde su sitio oficial.

## 3.2. Lenguaje de programación

Unity ofrece tres lenguajes para programar, estos son UnityScript (un javascript con ligeras modificaciones), Boo y C#, pudiéndose utilizar todos a la vez.

El lenguaje elegido ha sido C# por varios motivos. El principal motivo es que ya conocía el lenguaje, pero también ha influido el hecho de que es más fácil de depurar que un lenguaje de tipado dinámico como es UnityScript. Si esto fuera poco, la mayoría de la documentación está desarrollada en C# o UnityScript por lo que Boo quedaba descartado.

También se consulto con diferentes personas que trabajan desarrollando juegos con Unity y la respuesta de todos fue unánime, la mejor opción es C#.

C# (pronunciado *si sharp* en inglés) es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET, que después fue aprobado como un estándar por la ECMA (ECMA-334) e ISO (ISO/IEC 23270). C# es uno de los lenguajes de programación diseñados para la infraestructura de lenguaje común.

Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma .NET, similar al de Java, aunque incluye mejoras derivadas de otros lenguajes.

## 3.3. Introducción a Unity

Los elementos básicos en Unity se llaman GameObjects. La mayoría de elementos del juego dependen de estos objetos o están enlazados a uno. Desde los diferentes scripts que creemos para dar funcionalidad hasta componentes físicos, si queremos que estén incluidos en el juego deberemos asociarlo a un GameObject para que Unity lo tenga en cuenta.

Todos estos GameObjects se agrupan dentro de una Scene. Una Scene es un entorno donde se desarrolla una parte del juego. En este proyecto existen una Scene por cada nivel o escenario así como para los menús principales.

Como hemos comentado dentro de la Scene se encuentran diferentes GameObjects, que pueden ser personajes del juego, enemigos , la cámara, iluminación o cualquier otro objeto necesario. Cualquier componente que queramos tener en la escena deberá estar dentro de un GameObject, siendo un componente asociado a este. Algunos de los componentes más importantes son el *Renderer*, que se encarga de que el objeto sea visible, dándole una forma y un color o textura, el *Rigidbody* y el *Collider*, que gestionan las colisiones con otros elementos y las características de la física simulada por Unity, el *Camera*, que es, como indica su nombre, el encargado de renderizar

la escena de acuerdo a su configuración y el componente *Script* que permite asociar el comportamiento de un script con un GameObject.

Todos los atributos de los componentes se pueden editar tanto por código como desde el editor en tiempo de ejecución. Esto es especialmente interesante en los atributos de los scripts, los cuales si se declaran públicos pueden editarse para testear de forma sencilla sin tener que editar el código cada vez.

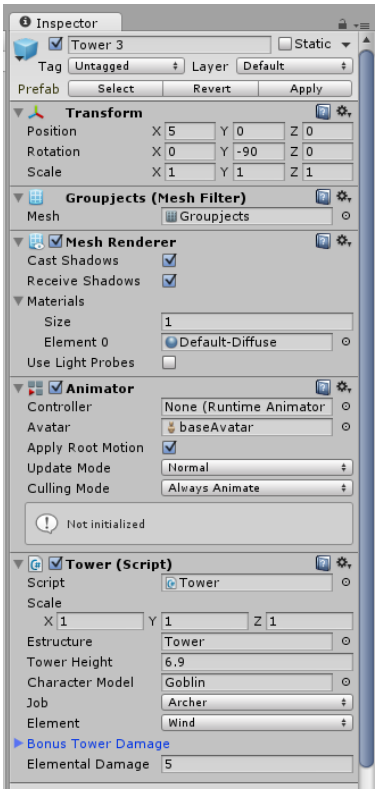



**Ilustración 11: Inspector de GameObject**

Como podemos ver todos los compontes pueden habilitarse o deshabilitarse excepto uno. El componente *Transform* viene por defecto en todos los GameObjects y decide la posición, rotación y escala que tendrá el objeto.

El ciclo de vida que utiliza Unity para tratar el comportamiento de los GameObjects está basado en el ciclo de vida clásico de los motores gráficos. Es decir primero inicializa los objetos por primera vez, una vez sucede esto entra en un bucle hasta que es destruido. dentro de este bucle primero realiza las actualizaciones físicas, luego realiza actualizaciones de código o animaciones, por último renderiza el objeto y vuelta a empezar hasta que se destruye.

Unity ofrece unos métodos que son ejecutados cada vez que se pasa por estas etapas. De esta forma reescribiendo los métodos manejadores somos capaces de controlar los objetos durante todo su ciclo.  Para poder acceder a estos métodos debemos hacer uso de *MonoBehaviour*.

Esta clase es la base del motor. Cualquier clase que quiera acceder a ciertas funciones básicas del motor debe heredar de ésta. Esto permite acceder a métodos como el *Start()*, que se ejecuta la primera vez que el *script*, o el *Update()*, que se ejecuta continuamente mientras el *script* esté activo, una vez cada *frame*.

En la siguiente imagen podemos ver el ciclo completo de vida con las funciones a utilizar desde MonoBehaviour.

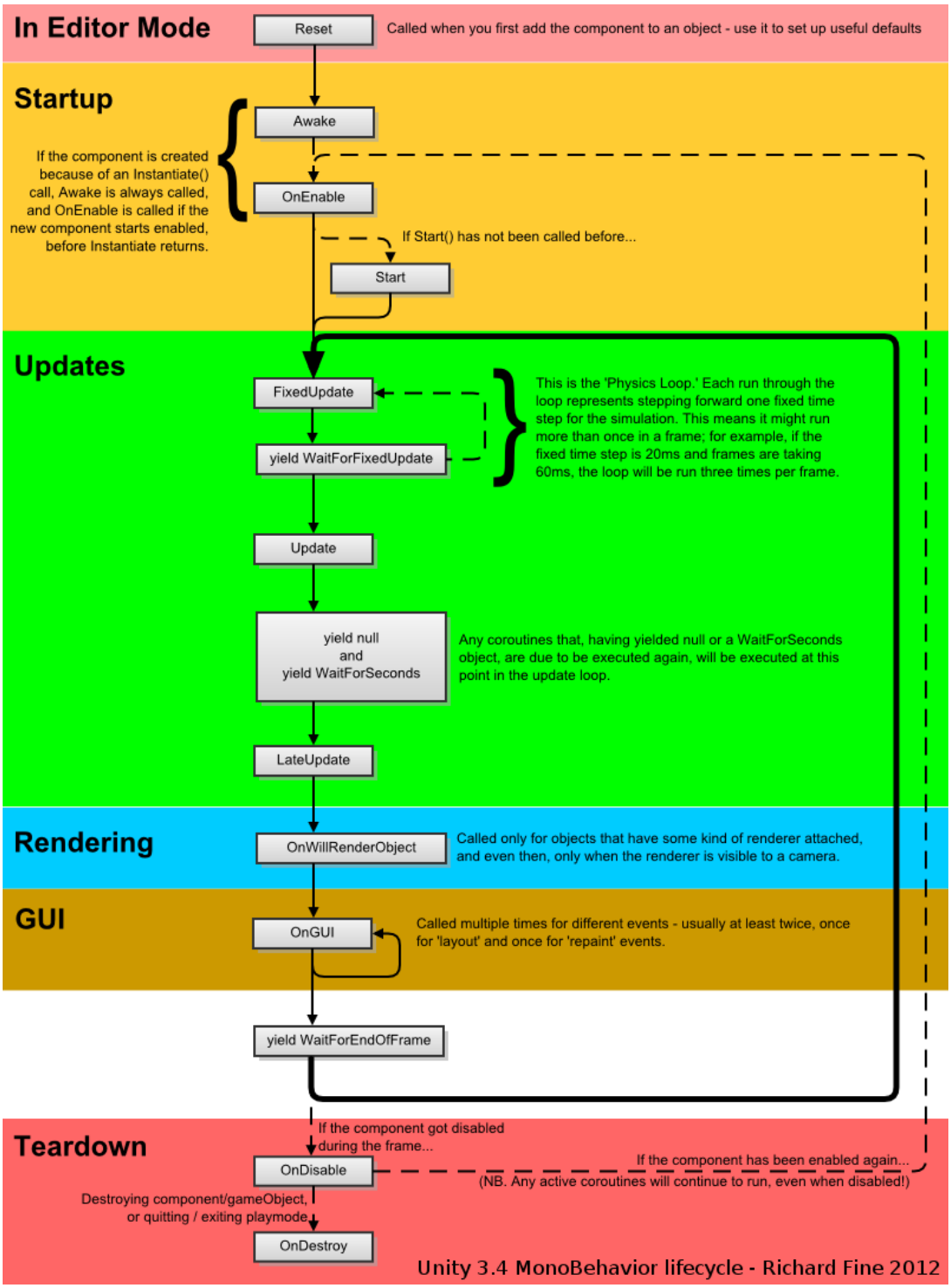



**Ilustración 12: Ciclo de vida de Unity, por Richard Fine**

# 4. Análisis

## 4.1. Planificación

En el diagrama de Gantt se puede observar la planificación del proyecto.

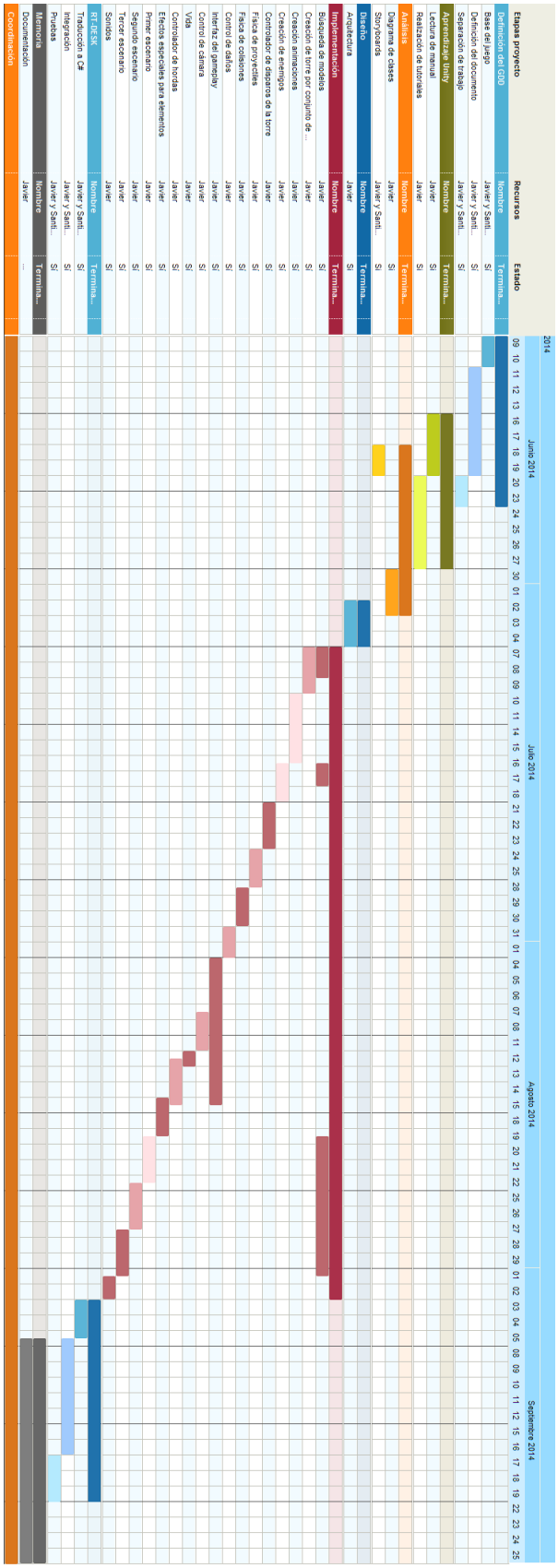



Ilustración 13: Diagrama de Gantt

El diagrama de Gantt abarca todas las fases del proyecto, desde la definición del GDD hasta la creación de la memoria. Este diagrama se ha creado a posteriori, por lo que es el resultado final de los tiempos del proyecto. Como se puede ver hay varias fases que se solapan. Sobretodo cuando son fases comunes con mi compañero. Puesto que las reuniones para desarrollar elementos comunes no duraban todo el día ni estábamos todos los días seguidos, se aprovecho para ir avanzando en otros aspectos.

Como se puede observar la fase más grande es la de implementación, esta es la que ha supuesto la mayor parte del trabajo.

Otro detalle importante es la fase de coordinación que ha durado durante todo el proyecto. Esto es debido a que durante todo el proceso hemos tenido reuniones y gastado horas en coordinar nuestros distintos proyectos (Gameplay y Editor). Es difícil saber cuántas horas a la semana hemos perdido en coordinación pues las reuniones se realizaban cuando surgían dudas.

## 4.2. Análisis funcional: Navegabilidad

En este punto vamos a describir los escenarios del juego completo (Proyecto Gameplay creado por mi y Editor creado por Santiago) así como la navegabilidad entre las distintas escenas.

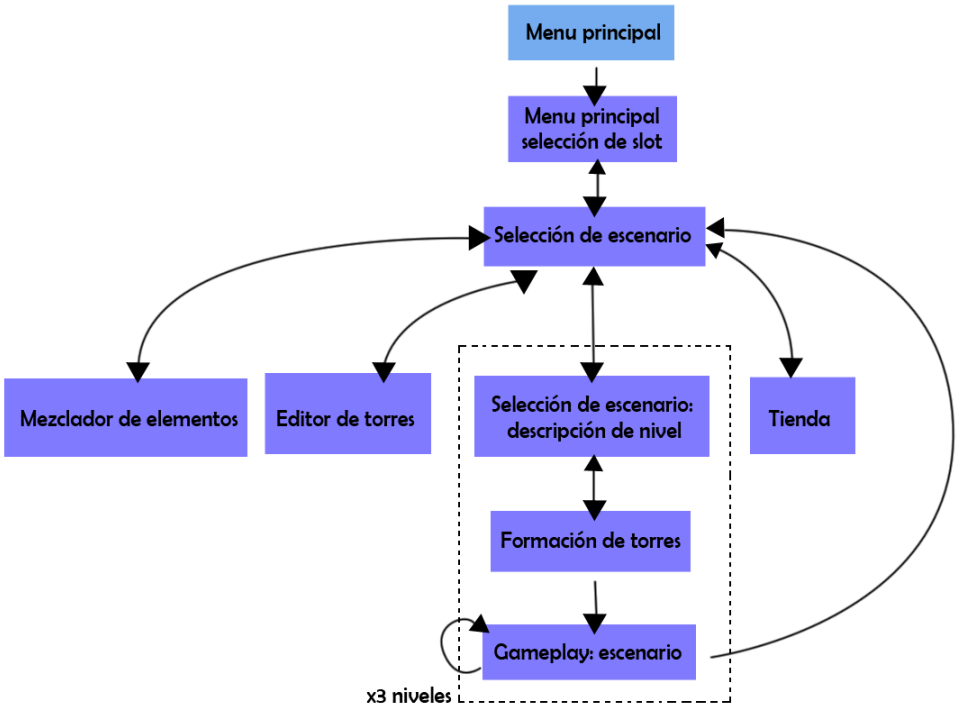



Ilustración 14: Diagrama de flujo

Como podemos ver en el siguiente grafo el punto de entrada al juego es el menú principal. Desde esta escena podemos ver la intro del juego así como entrar a la selección de slots, ver los créditos o salir completamente del juego.


Ilustración 15: Menú principal

A continuación pasamos a la selección de slot donde podremos elegir una partida guardada y cargar todos los datos acumulados en la persistencia.

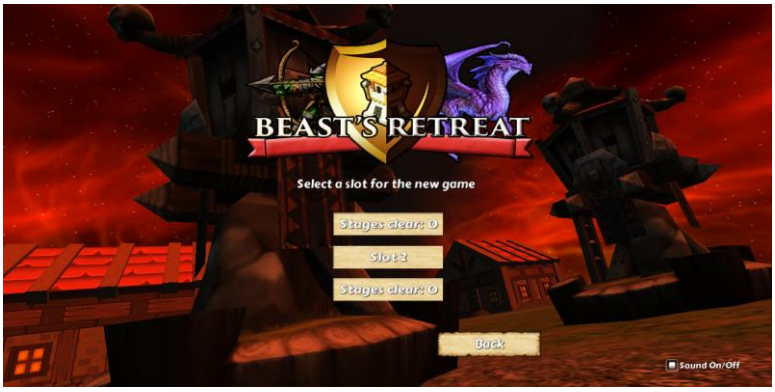



Ilustración 16: Selección de slot

Una vez se ha elegido un slot entramos en la selección de escenario. Este es el punto central del juego, a partir de aquí se puede acceder a todas las partes del juego: a todos los niveles y editores.


Ilustración 17: Selección de escenario

Para jugar una partida debemos pulsar en alguna de las pantallas que tengamos desbloqueadas, en ese momento se abrirá la descripción del escenario en concreto.


**Ilustración 18: Descripción de nivel**

Cuando le demos a Play nos llevará a seleccionar nuestra base o torres a utilizar.


**Ilustración 19: Formación de torres**

Una vez seleccionadas las torres que queramos utilizar iremos al Gameplay y a la escena que pertenezca a la partida en concreto.


**Ilustración 20: Gameplay**

De esta partida podemos volver a cargar la misma escena de Gameplay o volver al menú de selección de escenario.

En el menú de tienda podemos comprar varios objetos que nos servirán para montar nuestras torres o incluso fabricar otros objetos. Hay tanto objetos individuales como ofertas especiales.


**Ilustración 21: Tienda**

También tenemos la sección de fabricación de elementos, donde a partir de varios objetos podemos crear otros por combinación.


**Ilustración 22: Mezclador**

Por último tenemos el creador de torres mediante el cual podemos formar torres a partir de los distintos elementos fabricados. Estas torres serán las que luego utilicemos en las partidas.

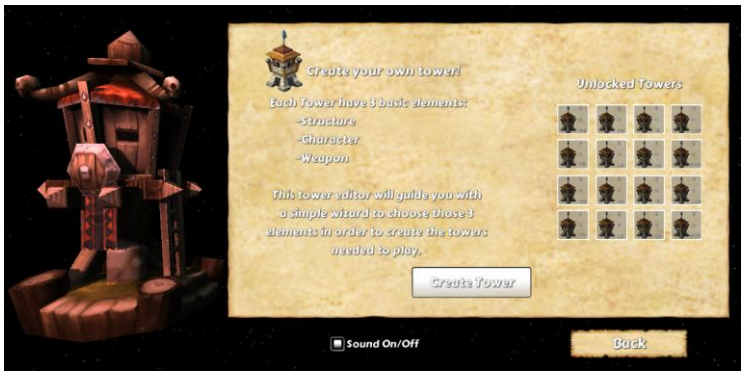



**Ilustración 23: Creador de torres**

## 4.3. Análisis de comportamiento: Máquinas de estado deterministas

Para determinar el comportamiento de los personajes es necesario crear unas máquinas de estado con todos los posibles estados en los que puedan estar.

Los diagramas de máquina de estado para nuestros personajes principales, es decir, los personajes de las torres es el siguiente:

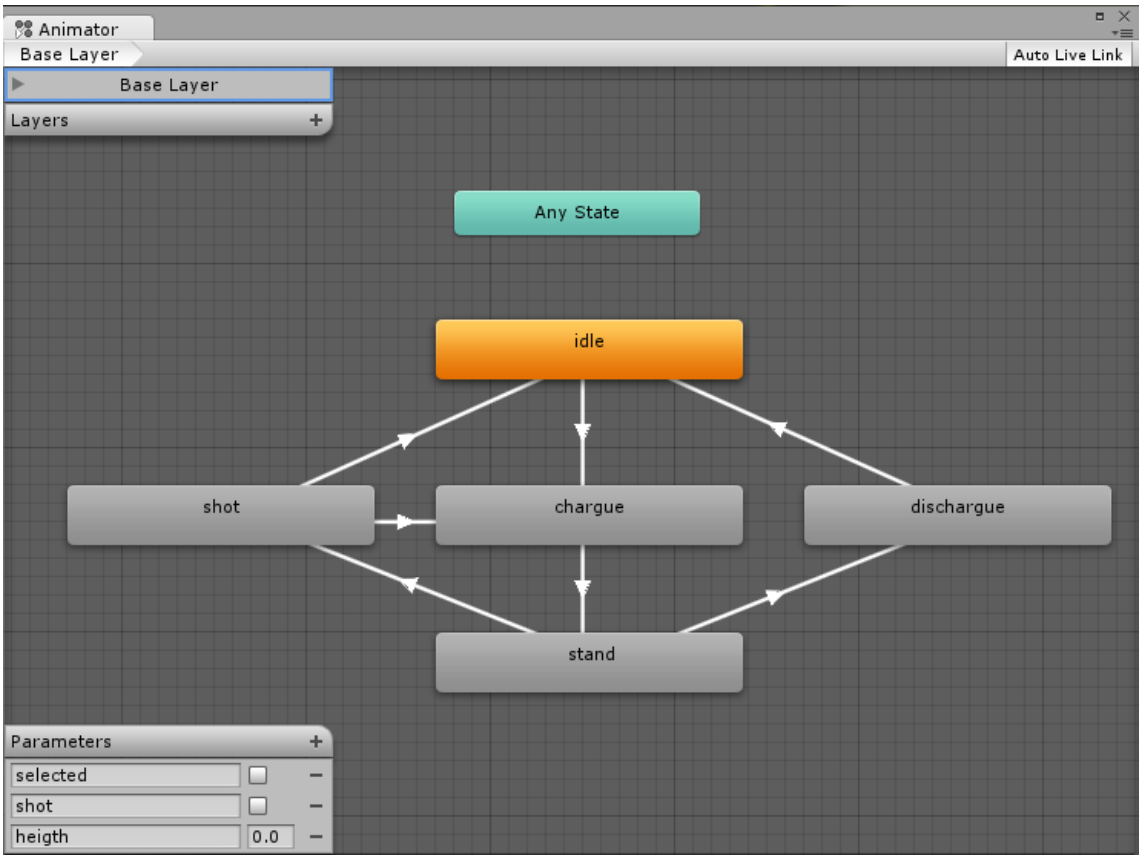



Ilustración 24: Máquina de estados para torres

Como podemos ver, el estado inicial es *idle* durante el cual el personaje se mantendrá quieto esperando que se le seleccione. Es decir, cuando el parámetro *selected* esté activo se cambiará de estado.

De ahí pasara al estado *chargue*, donde cargará el arma, de este estado saldrá en función del tiempo de carga.

Una vez haya cargado quedará en un estado stand durante el cual se podrá apuntar mediante el parámetro *heigth*. En este punto hay dos opciones, o bien disparar mediante el *trigger shot* y pasar a dicho estado o descargar el arma mediante deseleccionando el parámetro *selected*.

El estado *dischargue* volverá a *idle* cuando acabe la animación de descarga y *shot* hará lo mismo volviendo a *chargue* en función de si *selected* sigue activado o a idle en caso contrario.

Para los diferentes enemigos tenemos otra máquina de estados determinista para su comportamiento.

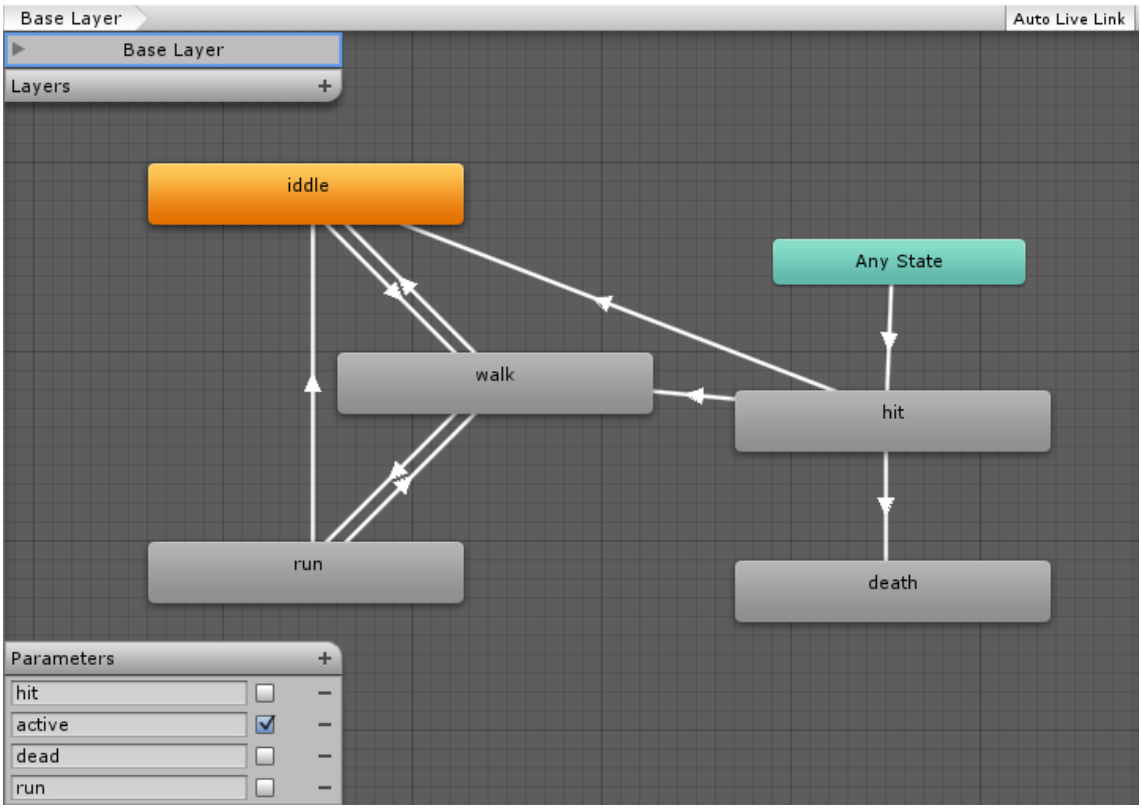



**Ilustración 25: Máquina de estados para enemigos**

Como podemos ver esta máquina empieza también con el estado *idle*. El cual pasará a *walk* en caso de estar activo el parámetro active.

En el estado *walk* los enemigos empezarán a andar. En caso de que se active el parámetro *run* entonces se pasará al estado *run* donde en vez de andar correrán. En caso de que se desactive el parámetro active volvería a estado *idle*. De igual forma ocurre con *run*, o bien se desactiva y vuelve a *iddle* o bien deja de correr y vuelve a caminar.

Por otra parte tenemos los estados hit and *death*. En cualquier momento se puede pasar al estado hit desde cualquiera de los otros estados, solo debemos activar el *trigger hit*, una vez activo pasamos a estado hit donde como su nombre indica el enemigo habrá recibido un golpe. En caso de que reciba un golpe y se active el parámetro *dead* se pasará a este estado del cual ya no saldrá.

También podemos volver a los estados *walk* o *idle* una vez finalice el tiempo de hit y dependiendo del parámetro *active*.

## 4.4. Análisis estructural: Diagrama de clases

Para definir las clases que necesitaremos se ha creado un diagrama de clases en la cual podemos ver todos los scripts que se han creado y su relación entre ellos.

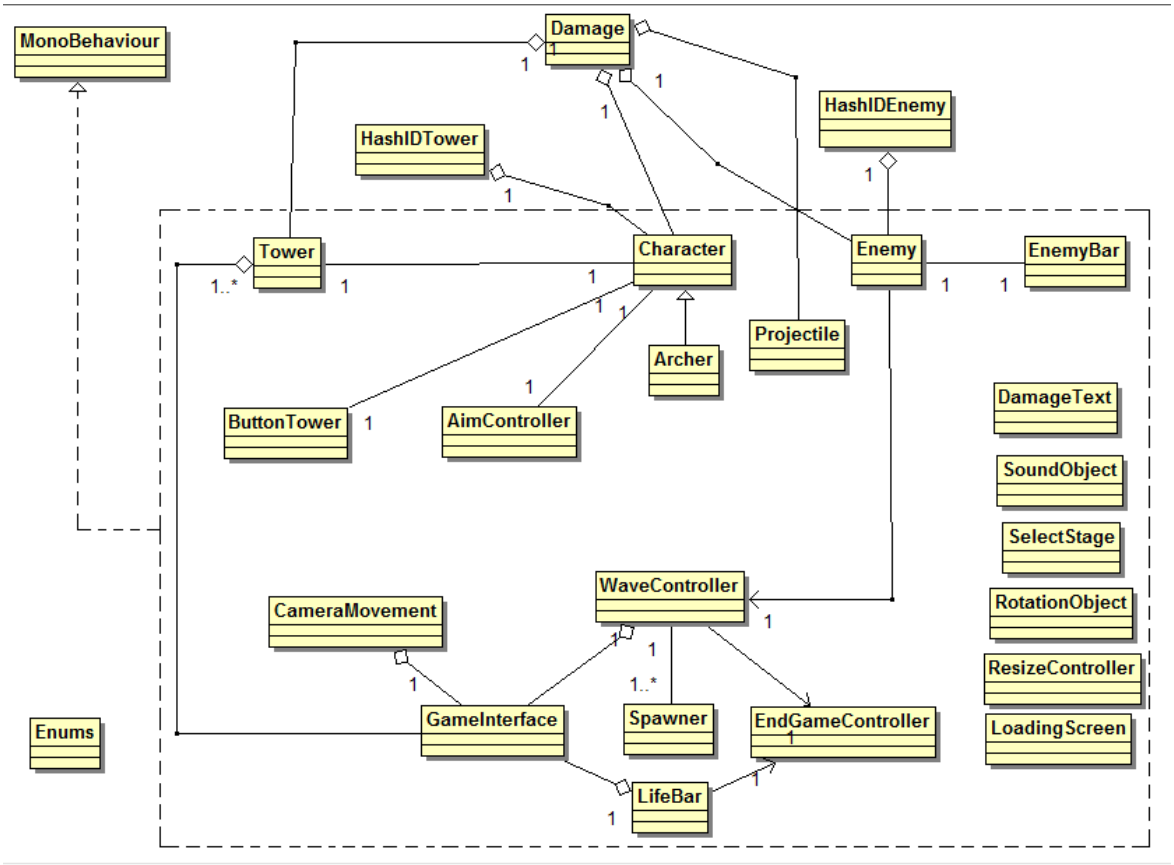



Ilustración 26: Diagrama de clases

Es muy importante comentar que todos estos scripts son utilizados desde GameObjects creados en Unity. De esta forma para poder hacer uso del motor y de diferentes funcionas, todas las clases que aparecen dentro del recuadro con líneas discontinuas heredan de la clase MonoBehaviour.

Los objetos que no heredan de MonoBehaviour son: Enums, HashIDTower/Enemy y Damage, y son clases diseñadas para contener datos por lo cual no hace falta que tengan funciones del núcleo de Unity.

Como podemos ver en el UML hay varios grupos de objetos diferenciados.

Por una parte tenemos los scripts para el comportamiento y control de la interfaz y las oleadas, por otro el control de la torre y los enemigos y por último una serie de clases dedicadas con control del flujo del juego.

## 4.5. Coordinación

Para coordinar este proyecto con el de mi compañero deberían integrarse los escenarios aquí creados con el editor de torres y toda la interfaz vista en la navegabilidad. Mi compañero Santiago Martínez Gómez realizó esto en su proyecto Videojuego Beast's Retreat en Unity con C# integrando RT-DESK: Editor. Para coordinar ambos proyectos se deberá pasar la información de las torres a utilizar a la escena concreta que se quiera jugar. Una vez acabada la partida se volverá al seleccionador de niveles.

Para lograr esta coordinación se han realizado a lo largo de proyecto múltiples reuniones para poner cosas en común. La primera de todas fue para detallar el GDD que se puede ver en el anexo. Una vez realizado esto se han ido implementando las características del juego en paralelo guiándonos en el diseño creado.

Para la integración de nuevas funcionalidades nos decidimos por crear un repositorio de versiones con GitHub al cual íbamos subiendo las nuevas versiones con funcionalidad probada. La metodología empleada para esto era la siguiente:

- Se implementa una funcionalidad o característica de forma local.
- Una vez hechas las pruebas funcionales se procede a comprobar si hay una nueva versión en el repositorio y se descarga en caso afirmativo.
- Se realizan pruebas para determinar si la integración de la nueva característica no produce errores con el resultado final
- Se sube la nueva versión con nueva funcionalidad y se pasa a implementar una nueva funcionalidad

De esta forma aun trabajando por separado, siempre teníamos una versión funcional del producto en el repositorio virtual. Mientras todo lo que subamos haya sido probado no hay problemas con la integración.

# 5. Diseño

## 5.1. Arquitectura

Como ya hemos visto en el análisis, los scripts utilizados están estructurados para poder encajar con la arquitectura utilizada por Unity. Es decir, que deben heredar de la clase MonoBehaviour para poder ser integrados como un componente dentro de los GameObjects que conforman la escena.

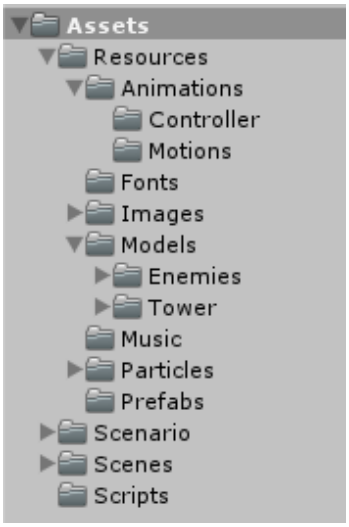



Ilustración 27: Estructura de carpetas

La estructura de carpetas utilizada nos servirá para poder entender mejor cual es la arquitectura utilizada.

Como podemos ver tenemos los diferentes recursos a utilizar separados en animaciones ( donde se encuentran tanto las animaciones como las maquinas de estado), las fuentes utilizadas, imágenes, modelos 3d, sonidos, sistemas de partículas y *prefabs*. Esta ultima carpeta es la más interesante pues es donde se encuentran todos los GameObjects predefinidos que luego se utilizarán durante la partida.

Son estos prefabs los que muestran la arquitectura que se ha utilizado pues en ellos podemos ver cada objeto que componentes tiene. Tanto los scripts definidos en el análisis como otros componentes de vital importancia y sus relaciones.


Ilustración 28: Lista de prefabs

Como podemos ver en la imagen, se han creado varios prefabs los cuales vamos a relacionar con el UML visto anteriormente.

Empezando por el *prefab Tower*, este estaría relacionado con la clase *Tower* y de hecho contiene dicho script. Pero no solo eso, tiene modelos asociados, transformación y una vez entra en ejecución el script *Tower* generará un personaje y lo integrará dentro de el mismo. Este personaje será un *prefab Goblin* o *Knight*, a los cuales se les asociará un script que herede de *Character* (Ahora mismo solo esta creada la subclase *Archer*, pero en un futura ampliación se añadirían más tipos de personajes) así como también se añadirían los scripts *AimController* y *GameButton*. Y aunque *character* no contiene ninguna relación con la clase proyectil sí que contiene un *prefab* con

dicho script. Según la clase se tendrá un tipo de *prefab* u otro que se utilizará como proyectil (en este caso *Arrow*).

Los diferentes *prefabs* de enemigos (*Troll*, *Golem*, *Zombie*, *Demons*...) estarían relacionados con el script *Enemy y EnemyBar ,* y estarían conectados con WaveController. Este WaveController tiene un script con el mismo nombre y tiene referencias a diferentes instancias de Wave (con script Spawner) que habría en el escenario.


**Ilustración 29: Inspector de componentes para un prefab**

Como podemos ver en la imagen de la izquierda. Dentro de un Prefab o GameObject predefinido tenemos varios componentes. La transformación para determinar su lugar y escala, un animador que dota de máquina de estados, diferentes scripts que son necesarios, sonidos, cuerpos físicos para controlar la física...

De esta forma nos podemos hacer una idea de la arquitectura utilizada en Unity y como todos los componentes se juntan dentro de GameObjects que podemos controlar con los scripts que les asociamos.

# 6. Implementación

## 6.1. Interfaz del gameplay

Desde este *GameObject* se controla prácticamente toda la interfaz del *Gameplay*.

Este *GameObject* está formado principalmente por un script *GameInterface* y un *AudioSource* asociado con un script *SoundObject*. También es importante que puesto que solo habrá un objeto de este tipo se ha decidido añadirle un *tag* especial llamado "*game interface*" para que los objetos que lo necesiten puedan acceder a él.

Esta clase contiene una referencia a varias torres como ya hemos visto en el UML. De esta forma al tener una relación con las torres de nuestra base podemos acceder a sus botones así como también tendremos referencias a otros objetos como el WaveController y el CameraMovement.

Estas referencias son importantes, las dos primeras para poder manejar el control los objetos de interfaz interactivos desde la misma clase y el segundo para poder llamar a una funcionalidad de la cámara. Todas estas relaciones se definirán durante la etapa Start del ciclo de vida de la interfaz.

Puesto que los muchos elementos son dinámicos y cambian su apariencia para simular relojes no podíamos utilizar los botones predefinidos por Unity por lo cual se ha tenido que implementar el control de los botones de forma manual. Para esto, durante la etapa Update del ciclo de vida se comprueba para cada elemento interactivo que el rectángulo en el cual se contiene el botón tenga el ratón encima y que este presionado el botón izquierdo del ratón. Si estás condiciones se cumplen es que se ha pulsado este botón y se llamará al método encargado de realizar cada función. Para evitar que se puedan pulsar botones mientras el juego esta pausado o hay un cambio de cámara también se comprueba que no se esté en movimiento.

```
if (rectangleChangeTowersButton.Contains (Input.mousePosition) && Input.GetMouseButtonDown (0) && !moving) {
        ChangeTowers ();
}
```

Una vez controlados estos botones se comprueba si el ratón ha entrado en la posición de algún botón para hacer sonar un sonido que indique que ese objeto es interactivo.

Por último es importante comentar que utilizando la clase *ResizeController* podemos saber mediante su atributo estático *isResizing* si el tamaño de pantalla ha cambiado y de esta forma recalcular la posición de los rectángulos. Esta clase está dentro del a primera escena y no desaparece entre cambios de escena y en caso de haber un cambio de tamaño cambia su el valor de su atributo.

Cada uno de los botones del juego tiene su comportamiento en esta clase.

Ilustración 30: Botón de sonido

El botón para el sonido recopilará todos los AudioSources de la escena y los pondrá el sonido a volumen 1 o 0 en función de si queremos sonido o no. También se almacenará este valor en una variable para que todos los objetos nuevos que aparezcan puedan acceder a este valor estático y saber si tienen que empezar teniendo sonido o no. Esto es gracias a la clase SoundObject que tienen asociados todos los objetos que producen algún sonido.

Ilustración 31: Botón de pausa

El botón de pause actuará sobre el tiempo del juego parando la velocidad de este y así evitando que cualquier avance en el tiempo de cualquier objeto. También almacenará en una variable si el juego está en pausa o no. Esta variable será utilizada entre otras cosas para saber cuándo mostrar o no el menú de pausa.

Ilustración 32: Botón de selección

Los botones de selección de torre desactivarán la anterior torre activa y activarán la torre a la cual pertenece dicho botón así como se guardarán una referencia a la actual torre activa. De esta forma serán los propios script Tower las que se encarguen de cambiar los estados de las maquinas deterministas, pero es la GameInterface la que lleva el control de activación de estas.

Ilustración 33: Botón de cambio de torres

De la misma manera ocurre con el botón para cambiar de vista y por tanto de fila de torres. Hay que actualizar la torre que se va a utilizar y además llamar a la cámara para que realice un movimiento para situarse en la nueva vista. Durante este movimiento el juego deberá pausarse.


Ilustración 34: Botones de navegabilidad

Otros botones que también son controlados desde aquí con los botones del menú de pausa. Estos botones no ha hecho falta controlarlos de forma manual pues al ser simplemente imágenes fijas que no cambian ni se animan puedan ser utilizadas mediante el uso de *Button* de Unity. La función de estos métodos es bastante simple, uno quita la pausa del juego volviendo a la normalidad de forma opuesta al método Pause que ya hemos comentado antes. Los

otros dos lo único que tienen que hacer es resetear ciertas variables estáticas de la clase y cargar un nivel. En el caso de *restart* cargarán el mismo nivel en el que estamos y en el caso de *quit* cargarán el nivel de selección de escenario.

El último botón que queda es el WaveController, pero este no tiene su funcionalidad aquí. Este cambio se debe a que GameController no contiene ninguna información sobre las hordas. El resto de comportamientos tienen su funcionalidad aquí o bien porque necesitan actuar sobre las torres y saber cuál es la actual torre activa o bien porque necesitan acceder a un cambio de nivel y por tanto resetear los valores estáticos. Aún así la clase contiene una referencia al rectángulo que ocupa este botón.

Estos rectángulos de los cuales hemos hablado son importantes pues al tener todos los botones controlados en una misma clase podemos tener un método que compruebe que el ratón no está encima de ningún elemento interactivo. Esto es de especial importancia para evitar casos en los que el usuario haga click encima de un botón pero también esté activo el *AimController* de alguna torre, así sabemos que tenemos que accionar el comportamiento del botón pero no empezar a cagar el disparo.

Para poder visualizar todos estos elementos se ha utilizado el método *OnGUI* donde se llama a los distintos métodos estáticos de la clase *GUI* para dibujar texturas entre otros elementos.

Para el botón de cambio de torres se ha utilizado un grupo dentro del cual dibujaremos una textura y otra en función de si estamos moviendo la cámara o no. De esta manera conseguimos que el botón aparezca deshabilitado mientras efectúa el cambio de torres.

Para dibujar los botones de selección de torre recorremos las referencias para acceder a las torres adecuadas según el punto de vista y llamamos al método *RenderButton* que tiene la clase *ButtonTower*. Es importante llamar a estos métodos desde esta clase porque de ser llamados en su propia clase se sobrepondrían los botones de ambas filas.

Para dibujar los botones de sonido es exactamente igual que el de cambiar torres pero cambiando la variable  para saber si el sonido esta activado o no. Y para el botón de pausa es aún más simple pues no hace falta comprobar nada.

Por último en caso de haber pausado el juego mediante el botón de antes se dibujará un texto "Game paused" con una *label* y 3 botones puestos en horizontal mediante un *GUILayout* que serían *resume*, *restart* y *quit*.

## 6.2. Cámara

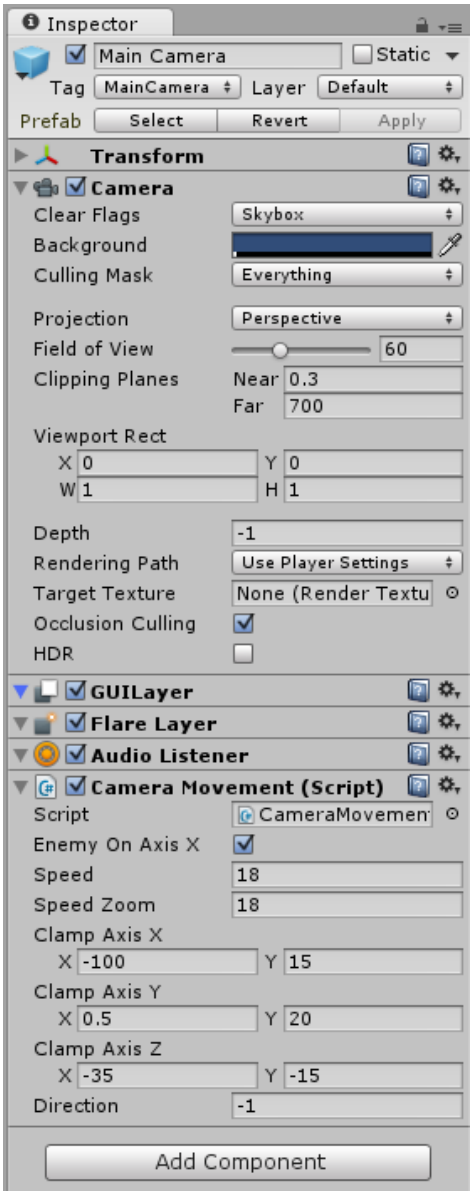



**Ilustración 35: Propiedades de la cámara**

Como podemos ver en la imagen de la izquierda la cámara necesita del tag MainCamera. Sin este tag no podríamos acceder a la cámara desde las otras clases, y esto nos vendrá bien en varias ocasiones.

A parte de esto es importante ver que tiene una proyección de tipo perspectiva un campo de visión de 60 grados y que es capaz de ver hasta una distancia de 700. Esto puede cambiar un poco dependiendo del escenario, pero en general está sobre ese límite.

El otro componente importante de este objeto es el script *CameraMovement.* En este script se define el movimiento que puede realizar el usuario para moverse alrededor del escenario así como un método especial que realiza una animación de la cámara para cambiar el punto de vista tal y como hemos comentado en el punto anterior.

Como ya sabemos los enemigos vienen por un camino hacia nuestra base en línea recta, por lo que los enemigos pueden venir tanto por el eje x como por z. Es por esto que se ha creado un atributo *bool* para poder elegir por cuál de los dos ejes vienen los enemigos. De esta manera para utilizar el script solo tenemos que añadirlo a la cámara, y poner los límites necesarios a los ejes y la velocidad de movimiento. Así cualquier diseñador de niveles podría dar movimiento a la cámara de una forma sencilla.

Los *Clamp* sobre el eje x mantienen la cámara entre dos posiciones a lo largo del eje por el cual vengan los enemigos, el del eje y sobre la altura y el del eje z sobre la profundidad.

La dirección indica el sentido hacia el cual se moverá dependiendo del movimiento del ratón.

Al empezar la etapa Start del ciclo de vida del script se obtendrá la posición de la cámara y en función del sentido de los enemigos se hará un cambio de ejes a la hora de almacenar esta posición.

Una vez hecho esto durante la etapa *Update* se comprobará si se ha pulsado el botón derecho del ratón, en caso de estar pulsado se actualizarán las posiciones x e y en función del *deltaTime*, es decir del tiempo transcurrido desde el último render, de la velocidad elegida y del valor devuelto por *Input.GetAxisRaw* de cada uno de los ejes. De esta manera conseguiremos el movimiento para el plano xy. Para el movimiento en profundidad utilizaremos la rueda del ratón, leeremos el valor de *Input.GetAxis* diciendo que lea la rueda de *scroll* del ratón y utilizaremos esto para avanzar o retroceder al igual que lo hemos hecho con los otros ejes. Por último asignaremos estos valores a la posición de la cámara.

Para realizar el movimiento de cambio de vista se utiliza un método iterador que devuelve un IEnumerator. Esto es un requisito para poder crear una corutina con la cual poder hacer una animación cuando tenemos el tiempo parado. Recordemos que durante el cambio de vista todos los objetos deben estar quietos.

La animación sería la siguiente. Primero se coge un punto que esté a la misma altura sobre el eje por donde vengan los enemigos. Luego se rotaria 90 grados sobre el eje x de la cámara tomando como centro el punto calculado anteriormente. Esto se hace dentro de un bucle de uno en uno mientras ejecutamos un wait para esperar un frame nuevo. Una vez hecho esto se aplica una rotación de 180 grados sobre el centro de la cámara en un bucle esperando un frame por cada 5 grados. Por último se vuelve a rotar partiendo del punto calculado al principio otros 90 grados de la misma forma. Se recolocan las variables de control y se quita la pausa.

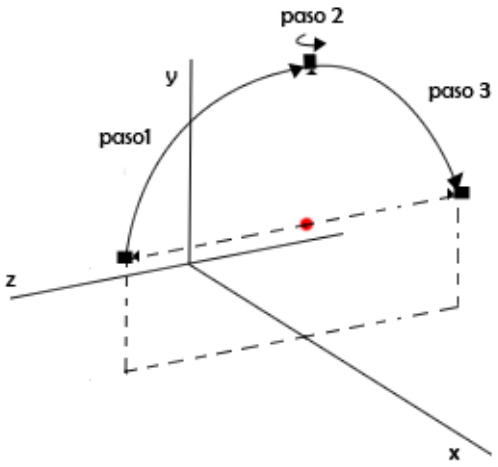



Ilustración 36: Movimiento cambio de vista

## 6.3. Torres

Las torres del juego se componen de tres componentes: estructura, personaje y arma.

La clase *Tower* contiene todos los componentes, la estructura estaría compuesta del modelo 3d de la estructura, el cual puede editarse seleccionando el *GameObject* del modelo 3d que creamos oportuno en el atributo *estructure*. A parte del modelo también será necesario saber cuál es el *bonus* que ofrece esa estructura, esto sería el atributo *bonusTowerDamage*.

En la etapa *Awake* se llamara a un método *CreateEstructure* donde se instanciará el modelo 3d y se asociará su transformada como hija del GameObject actual.

El personaje está representado por un modelo 3d y un script que lo controle. El modelo 3d se asocia mediante el atributo characterModel y el script será obtenido a través de un enumerado con las clases disponibles para facilitar su creación por parte de otra persona ajena a los detalles de su programación.

La creación del personaje se hace también desde la etapa *Awake* cuando se llama al método *CreateCharacter*. En este método se instancia el modelo del personaje a partir del *prefab* asignado, se estructura dentro de la transformada original en una posición situada a una altura que viene dada por *towerHeight* y se le asigna el script necesario para la clase y que hereda de *Character*. también se le añade un script para poder controlar al personaje (*AimController*) así como otro script para la selección de torre (*ButtonTower*). De esta forma *Character* contendrá ambos clases como se especifica en el UML.

El arma a utilizar estará también asociada al atributo *job* que contiene las clases disponibles, pero además de esto también habrá otro enumerado para poder asignarle un poder elemental a la arma y un atributo *elementalDamage* para decir cuánto daño adicional hace el elemento.

A parte de estas inicializaciones contiene varios métodos que sirven de interfaz con *GameInterface*. Estos métodos permiten habilitar y deshabilitar la torre así como obtener los botones de la torre sin tener necesidad de conocer la clase *Character* ni como está implementada.

### 6.3.1. Personajes

Para poder utilizar un modelo como personaje de nuestra torre debemos crear un prefab que contenga el modelo 3d. Pero no vale cualquier modelo 3d, necesita tener tanto la malla poligonal como una estructura de huesos o transformaciones. La mayoría de modelos que podemos encontrar por la asset store de Unity ya la llevan incorporada, pero de querer utilizar otros modelos o crearlos nosotros mismos deberemos asignar los huesos.

Una vez tenemos un personaje con malla poligonal y estructurado deberemos asignar ciertos tags que son necesarios para colocar las armas a los personajes. Estos tags son *leftHand*, *rightHand* y *leftShoulder*. Estos tags se asignaran a las transformadas que marcan la manos y el hombre respectivamente. Si hicieran falta más puntos para asignar armas solo deberíamos declararlos como atributos *protected* en *Character*, marcarlos con un tag identificativo en todos los modelos que queramos utilizar y luego buscarlos desde el script como se explicará en el siguiente punto.

Ilustración 37: Transformada leftHand

### 6.3.2. Clases

#### *6.3.2.1. Character*

Las clases a elegir son definidas por el arma que queramos utilizar. De esta forma si queremos utilizar un arco necesitaremos hacer uso de la clase *Archer*.

Estas clases heredan de la clase abstracta *Character*. Esta clase está relacionada con *HashIDTower*, *Animator*, *ButtonTower*, *AimController* y *Tower*.

Las dos primeras clases tienen su función para manejar la máquina de estados deterministas a través de los atributos que se comentaron anteriormente. La primera clase contiene los ids tanto de los estados como de los atributos de esta máquina. Al iniciarse la clase accede a ellos a través de la clase *Animator* y se guarda todos esos ids, que luego serán utilizados desde *Character* y sus subclases para cambiar o leer valores de la máquina de estados.

Las siguientes dos relaciones vienen especificadas desde su creación en la clase *Tower*, como ya hemos explicado antes, y nos servirán para controlar la torre así como mostrar el estado en el que se encuentra.

Además de estas relaciones contiene otros atributos que deberán ser sobrescritos en las clases herederas. Estos atributos son dos imágenes

necesarias para los botones de selección de torre. Según la clase se asignara un icono u otro. También tenemos un atributo *maxPower* que servirá para definir la fuerza máxima con la que se lanzará el proyectil (un humano con un arco no tiene la misma fuerza que una catapulta...). Otros atributos importantes son los diferentes GameObjects con partículas que se utilizan para representar los elementos. Por defecto se ofrecen unas partículas de tamaño medio pero si se necesitase obtener otros efectos se deberían reescribir en la subclase dentro del método oportuno. (No es lo mismo una flecha en llamas que una piedra en llamas...) Por último comentar que también se tendrá un atributo que enlazaremos con el prefab del proyectil a lanzar y otro método donde nos guardaremos la instancia del proyectil que tengamos en la mano. Esto se explicará mejor en el siguiente punto.

En la etapa *Awake* de esta clase, se recuperan las relaciones con la torre y con el animador mediante el uso de *GetComponentInParent* para la torre y *GetComponent* para el animador. Recordamos que este script se encuentra dentro de la estructura principal del *prefab Tower*, por lo que podemos acceder a componentes que están en el propio *GameObject* como a cualquiera que esté por arriba o por debajo. El componente *Animator* es un componente que aparece por defecto en todos los *GameObjects* por lo cual no ha hecho falta añadirlo, pero en caso de no tener un *AnimatorController* habría que añadir dicho componente.

Una vez hecho esto añade un componente de tipo script para añadir la clase *HashIDTower* y justo a continuación se llamará a un método abstracto que cargara el animador pertinente en el controlador de la animación. Este método abstracto deberá ser cargado de forma dinámica por las diferentes subclases. Estas subclases cargarán una máquina de estado específica para su clase. Esto es así porque Unity ofrece la oportunidad de crear maquinas de estado que heredan de otra máquina. De esta forma podemos cambiar las animaciones manteniendo los estados y puesto que todas las clases tendrán los mismos estados pero con distintas animaciones era la forma ideal de hacerlo.

Lo siguiente que hará será buscar utilizando los tags para asignar los *GameObjects* que contienen las transformaciones definidas en los prefabs de los modelos. Es decir para tener referencias a las dos manos y hombro del personaje.

A continuación se le dará valor al bonus de daño por raza en función del modelo que utilicemos y se llamará al método abstracto *LoadWeapons* que todas las clases herederas deberán sobrescribir para instanciar los modelos de las armas (arco, funda, proyectiles ....).

En la etapa *Start* se llamará al método *SetEffect* que seleccionará el efecto a añadir a los proyectiles según el elemento que hayamos decidido anteriormente. Estos efectos se explicarán más adelante.

Durante la etapa *Update* esta clase controlará la etapa en la que nos encontramos. Si nos encontrásemos en la etapa *stand,* entonces avisaríamos al *AimController* para decirle que debe mostrar la barra de fuerza. Así como también modificaríamos el valor *heigth* de la máquina de estados, con la cual apuntamos para que haga una interpolación de la animación, a partir del valor obtenido de dividir la rotación entre el ángulo máximo permitido del AimController.

Esta clase contiene otros métodos útiles para habilitar la selección de personaje a través de activar el atributo *selected* de la máquina de estados, así como también métodos para controlar los *timers* del *ButtonController*. Por último en esta clase deberíamos comentar que existe un método abstracto llamado *Shot* que deberán implementar todas las subclases.

#### *6.3.2.2. Subclases de Character*

Para esta versión del juego solo se ha creado una subclase de *Character* y por tanto solo hay un tipo de arma a utilizar. Aunque solo tengamos la subclase *Archer*, crear otras subclases seguiría exactamente el mismo proceso.

Dentro de cada subclase deberemos añadir atributos para poder referenciar los modelos de las armas. En nuestro caso necesitaremos arco, carcaj y montón de flechas.

Dentro del método *Awake* deberemos llamar a *base.Awake* para así ejecutar todo lo ya visto en la superclase. De esta forma se llamará al método LoadAnimator y LoadWeapons que en esta clase aparecen implementados.

El método *LoadAnimator* cargará el *AnimatorController* de esta clase que se encuentra en Resourses/animations/controllers y lo asignará haciendo uso de la propiedad *runtimeAnimatorController*. Esta es la única manera de cambiar el controlador de forma dinámica.

El método *LoadWeapons* por su parte cargará los modelos necesarios haciendo uso de *Resources.Load* y los almacenará en las variables mencionadas anteriormente. Una vez hecho esto las instanciará en distintas posiciones tomando como punto de origen las transformadas de los objetos *rightHand* y *leftShoulder* declaradas en la superclase. Es importante decir que hay que posicionar las armas de forma separada según el modelo porque los artistas que los han creado no han seguido una estructura estándar. De haber seguido todos una misma forma de trabajar no habría hecho falta hacer esta diferenciación según el modelo.

Una vez colocadas todas las armas, acaba el método *LoadWeapons* y con ello el *Awake* original y pasamos al *Awake* de la subclase. Aquí es donde se modificarán los atributos *maxPower* para adaptar la fuerza al arma así como se cargarán de recursos las imágenes de los iconos para los botones y los diferentes *prefabs* de elementos. Estas cargas desde código se deben a que Unity pierde las referencias al recargar la escena si están guardadas en el editor de scripts. Esto no pasa con los prefabs.

Para controlar las diferentes etapas propias de las subclases deberemos controlar: el disparo, el momento en el que se coge el proyectil, el momento en el que se empieza a cargar, el momento en el que empieza la animación de la arma. Esto se traduce en varios métodos que describiremos a continuación.

El método *Shot* se ejecuta en el momento que se para de mantener presionado el botón izquierdo del ratón. Esto es controlado por la clase *AimController* que en cuanto ocurra esto llamará al método *Shot* con los parámetros oportunos. Se ejecutará el método *StartClock* al principio de la llamada para así empezar el tiempo de recarga en el botón asociado. Después de esto llamará a *InstantiateEffect*, que instanciará el efecto de elemento sobre el proyectil. Tendremos que liberar la flecha de la estructura interna de la torre y soltarla al mundo exterior así como también desbloquear el cuerpo físico de la flecha que hasta ahora estaba en suspensión para evitar colisiones con otros objetos e interacciones de fuerzas. Mediante los parámetros de entrada se calcula la dirección en 3 dimensiones que debe seguir la flecha y se aplica una fuerza sobre el cuerpo físico para que así la flecha salga disparada. Por último se hace sonar un sonido asociado al proyectil y se activa el *trigger shot* en la máquina de estados, que hará que se ejecute la animación.

Cuando se ejecute la animación de disparar no solo hay que animar el movimiento del personaje, también el del arco por lo que hará falta un método que se ejecute en un determinado momento de la animación. Este momento puede verse en la siguiente imagen.

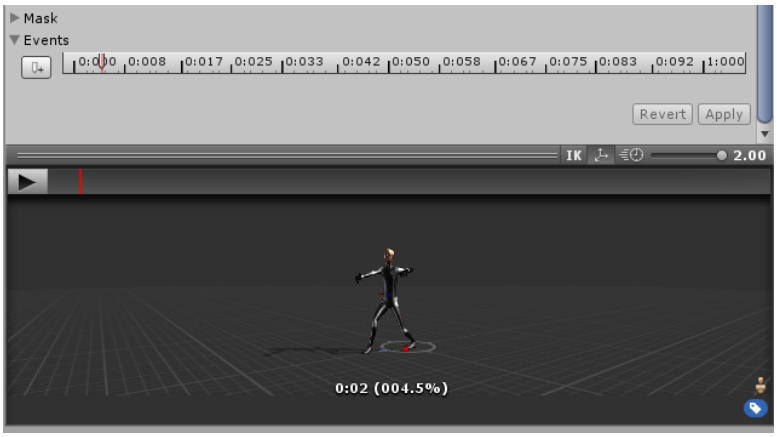



**Ilustración 38: Eventos durante la animación shot**

Unity permite hacer saltar un trigger cuando se pase por cierto frame de una animación de forma que asociamos un momento concreto con el método ChargueBow.

Este método al ejecutarse comprobará si se está en el estado de carga o descarga para así ejecutar la animación del arma en un sentido u otro. Es decir que el arco y la cuerda se tensen o se destensen.

El resto de métodos que faltan por comentar de esta clase siguen el mismo patrón de ejecución. Dependen del momento de la animación *chargue*

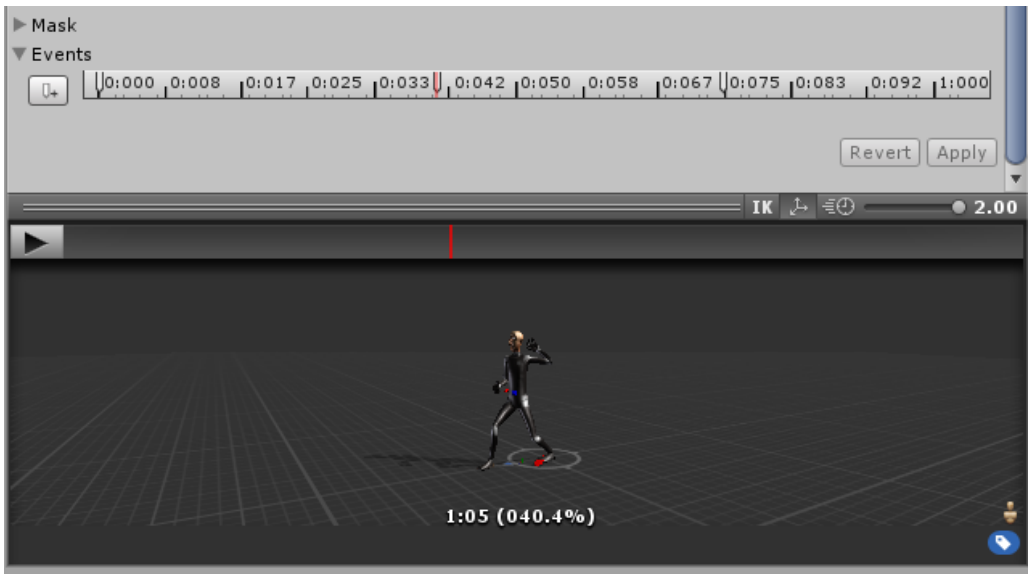



**Ilustración 39: Eventos en la animación chargue**

Como podemos ver en la imagen tenemos dos eventos enlazados, el primero de ellos con el método *Chargue*, el cual empieza el segundo tiempo de carga sobre el botón de la torre, el segundo método es *PickProyectil*.

Este método crea una instancia de flecha a partir del *prefab* almacenado el *projectilPrefab* y almacena dicha instancia en *projectilReady*. Se llama a *PrepareProyectil* para añadirle a la clase *Projectile* el daño total que sería capaz de infligir a un monstruo sin resistencia. Una vez hecho esto se coloca la flecha de forma que este cogido por un extremo en la mano de nuestro personaje y se traslada la transformación para que sea hija de mano y se mueva con ella durante el resto de animación. Todo esto es cuando el estado de la maquina es el de cargar, pero cuando debe descargar simplemente lo único que hace es destruir la instancia de flecha y así es como si guardara de nuevo la flecha en el carcaj.

#### 6.3.2.3. Animaciones por subclase

Como hemos podido ver, las animaciones tienen un papel clave en el manejo del personaje.

En concreto las animaciones para esta subclase se han realizado con 3DMaxStudio puesto que no se han encontrado animaciones para arquero en la asset store.

Para realizar las animaciones se ha empleado un modelo esquelético que viene por defecto con el programa. Se han tomado capturas de una serie de posiciones y así al colocarlas sobre una línea de tiempo y haciendo una interpolación de frames se consigue este resultado.

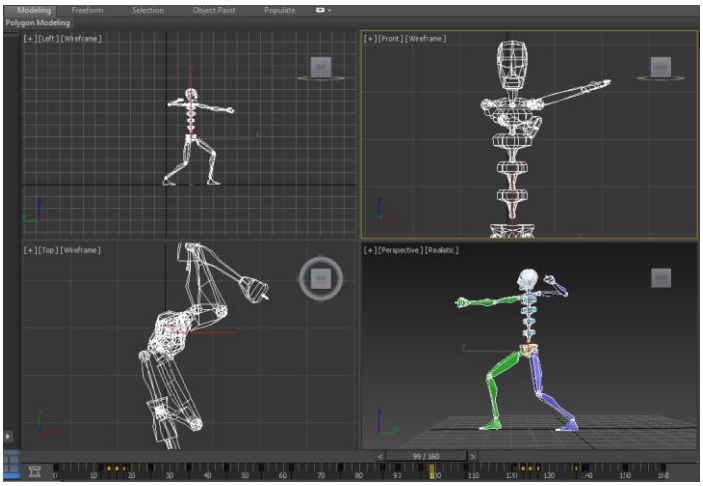


Ilustración 40: Animación creada con 3DStudioMax

Como podemos ver en la imagen, cada punto negro como de colores son distintas posiciones que han hecho falta para realizar las distintas animaciones del arquero.

Una parte importante de la animación ha sido apuntar. Para poder lograr este efecto en función de algo externo, como es la posición del ratón por la pantalla, se ha necesitado coger dos animaciones con movimientos opuestos y hacer una aproximación en función de este factor. Esto con Unity se consigue haciendo un estado mediante un árbol de mezclas como el que vemos.

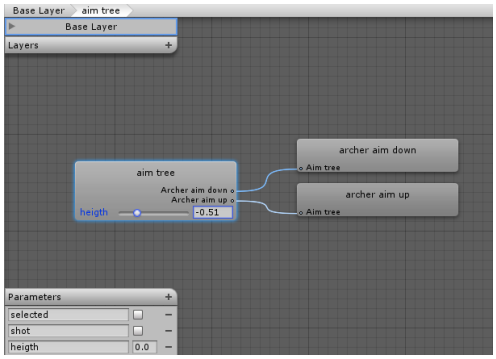



Ilustración 41: Subtree de la máquina de estados

En este árbol se introducen ambas animaciones opuestas y el resultado es una media entre ambas según un parámetro. Como podemos ver si cambiamos el valor de *heigth* podemos ver cuál sería el resultado al ejecutarse la animación. De esta forma podemos editar los valores de animación y ajustarlos hasta el resultado sea el correcto.

Estos ajustes no solo son necesarios durante el calibrado de los árboles de mezcla, son necesarios también para calibrar las transiciones entre las animaciones de un estado y otro.

Como podemos ver en la segunda imagen debemos calibrar la curva de forme que sea lo más suave posible y que coincida. Esto no siempre es fácil y necesita de muchos ajustes y pruebas para que quede correcto. Para ayudar con esta labor podemos ver el resultado de la transición en la pre visualización e ir frame a frame hasta que quede como más nos acomode.

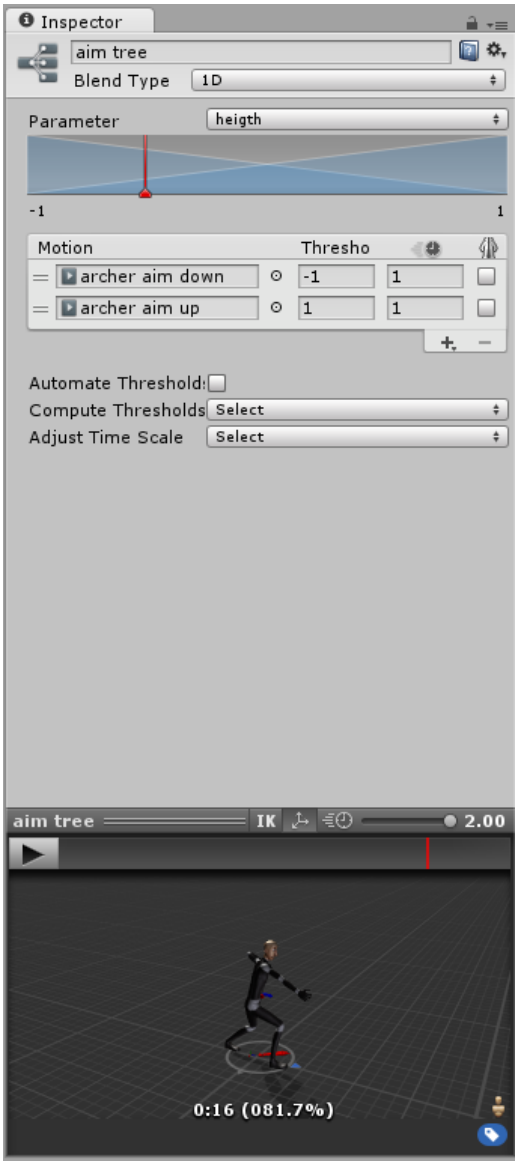



Ilustración 43: Inspector de un blend tree

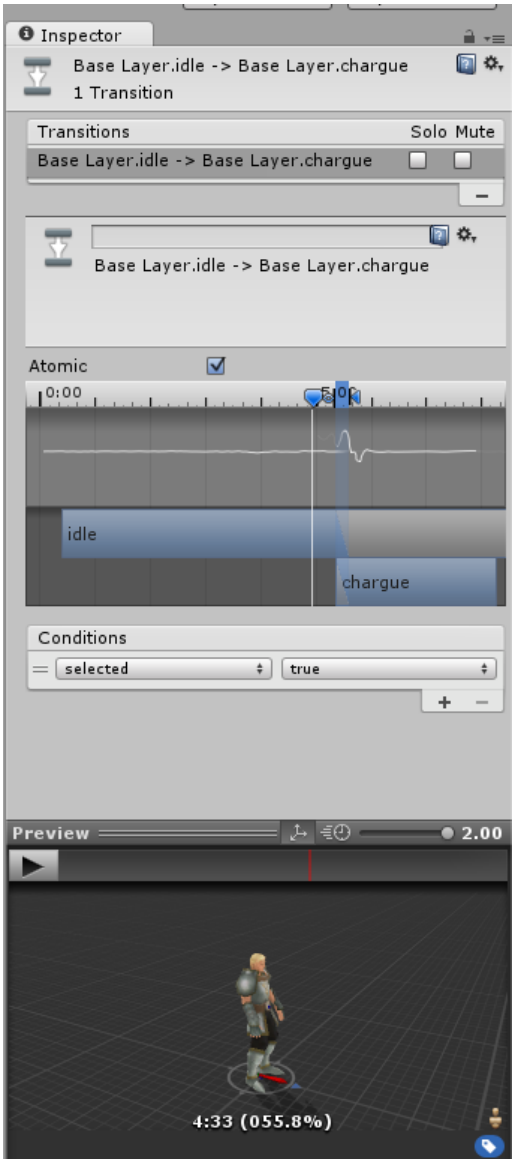



Ilustración 42: Inspector de una transición

### 6.3.3. Proyectiles

Los proyectiles deberán ser *prefabs* que contengan un modelo 3d así como un cuerpo físico y un controlador de colisiones así como un script de tipo *Projectile* para controlar su comportamiento al colisionar.

La clase *Projectile* tendrá su propio daño básico, es decir sin contar los bonus ni daños elementales y cuando se instancie será modificado por la clase *Character* para añadir el daño por bonus y el elemental como se ha comentado anteriormente. Además de esto también tendrá un atributo *element* de tipo enumerado donde según el elemento que se le asocie utilizará un color u otro para mostrar el daño por pantalla. A parte de esto también contará con un factor *critical* donde se especificará el bonus por crítico de ese proyectil.

Esta clase tiene 3 funciones básicas. La primera se produce en la etapa *Update* y consiste en crear un trazado de líneas con la trayectoria del proyectil cuando estemos en modo debug. Esto nos ayudará a seguir mejor el proyectil y poder depurar fallos.

La segunda función se ejecuta durante la etapa *FixedUpdate* y consiste en aplicar una rotación al proyectil cuando tenga velocidad para así calibrar la correcta rotación del proyectil. En el caso de flechas y lanzas el motor físico no controla correctamente la simulación de un lanzamiento de estas características por lo que mediante estas pequeñas rotaciones conseguimos que la punta de la flecha vaya siempre por delante. Esto se consigue almacenando la posición anterior y haciendo que mire a la posición actual mediante *LookAt*.

La tercera función es *OnCollisionEnter* la cual saltará cuando se produzca una colisión. En caso de colisión hay 2 posibilidades o bien a colisionado con el suelo de forma que dejaremos el proyectil quieto clavado en el suelo quitándole el colisionador y volviéndolo un objeto estático, o bien ha chocado con un enemigo y deberemos tratar el daño.

En caso de darle a un enemigo haremos como en el caso anterior y le quitaremos el colisionador y volveremos estático pero le cambiaremos la transformada padre para que entre dentro de la estructura del enemigo y se mueva con él como si de verdad estuviera clavado en el cuerpo del monstruo. Una vez hecho esto deberemos aplicar el daño al monstruo, es por esto que según si le hemos dado en la cabeza o en el resto del cuerpo aplicaremos un daño critico u otro normal mediante el uso del método *ReceiveDamage* del enemigo.

Por último eliminaremos todo rastro de partículas en el proyectil, una vez que choca el efecto se apaga.

Esto es todo lo que hay que comentar de la clase *Projectile* pero no es suficiente para poder tener un prefab adecuado. Un prefab que quiera ser utilizado como proyectil necesitará varios componentes más. Para ello vamos a explicar los elementos necesarios utilizando como ejemplo el *prefab Arrow.*

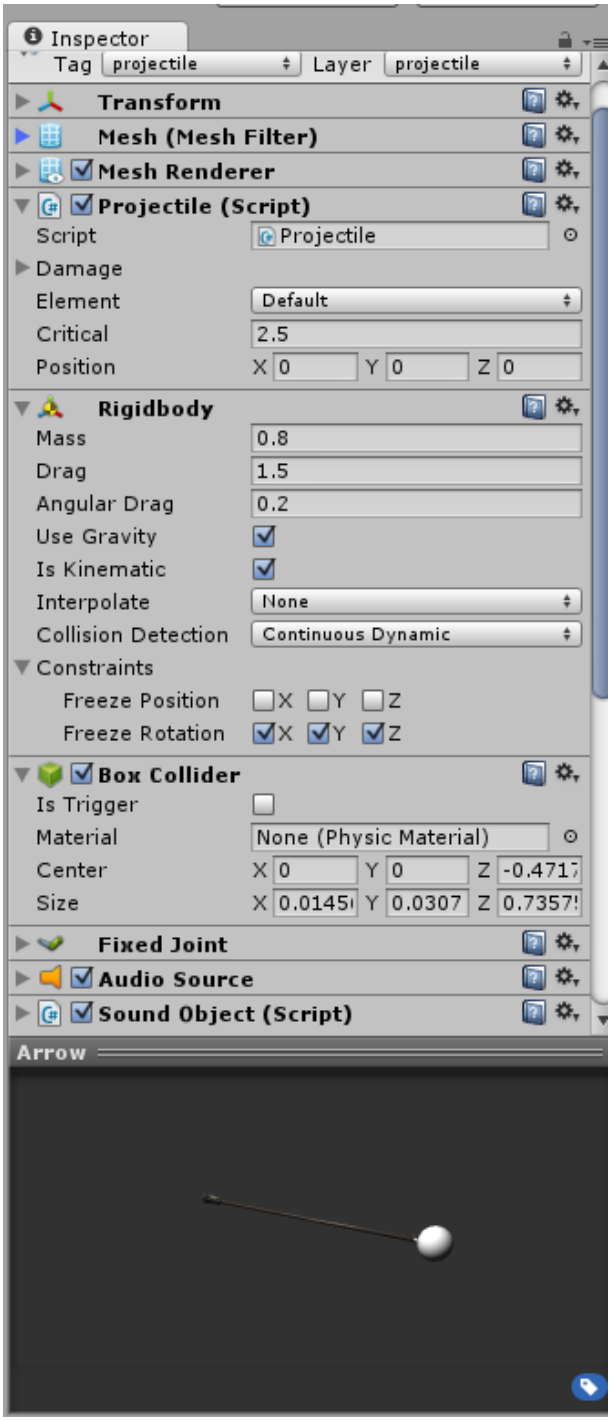



**Ilustración 44: Propiedades físicas de un proyectil**

Como podemos ver en la imagen todo proyectil necesitará un *Rigidbody* que determine el peso, como es afectado por las distintas fuerzas, su resistencia a ellas... Esto dependerá de cada proyectil y cambiará dependiendo de su física real.

Como podemos ver en la pre visualización , hay una esfera en la punta de la flecha, esta esfera no se renderiza pero contiene otro cuerpo físico que le da un mayor peso en la punta, de esta forma logramos una física mucho más realista, pues las flechas contienen más peso en la punta que en el resto del cuerpo.

Otro detalle importante es la detección de colisión dinámica, es importante que sea así para evitar posibles pérdidas de colisiones. Así como también es importante bloquear las rotaciones por física, puesto que las rotaciones serán controladas por el script como ya hemos comentado.

Puesto que la flecha contiene dos cuerpos físicos es importante que estén unidos para el motor físico, es por ello que necesitamos un *FixedJoint* que junte ambos *RigidBody*.

También vemos que contiene un *AudioSource* asociado, esto es para hacer sonar un sonido de flecha cuando se lanza, este sonido se ejecuta cuando en el método *Shot* como ya hemos comentado anteriormente.

Por último deberíamos comentar las colisiones de este objeto. Como hemos visto se le ha asociado un *BoxCollider* con forma de caja que cubrirá la flecha. Pero no solo esto, si vemos el *tag* y el *layer* veremos que son de tipo *projectile*. Esto es de suma importancia porque en el motor de física se ha configurado para que *layer projectil*e no colisione con objetos normales. Esto era necesario porque al cargar la flecha del carcaj al arco podía chocar con otros objetos haciendo que tomara otro rumbo y cambiando su velocidad.

**Ilustración 45: Box Collider de una flecha**

Podemos ver cuál es la configuración de la física en la siguiente imagen. Cabe destacar que los proyectiles no pueden chocar con objetos de la capa default ni con otros proyectiles. Así como los enemigos no pueden chocar entre ellos.

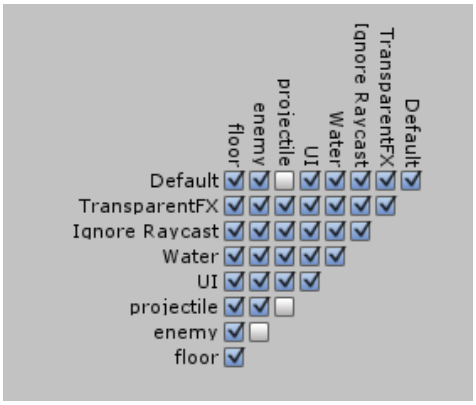



**Ilustración 46: Configuración de colisiones**

### 6.3.4. Controlador

Para controlar la torre se utilizan dos objetos: el AimController y el ButtonTower. El primero sirve para controlar la barra de fuerza con la que disparar y el segundo muestra el tiempo de recarga y permite seleccionar una torre.

#### *6.3.4.1. AimController*

Esta clase controlará la barra de fuerza y los disparos de las torres. Esta barra de fuerza solo se mostrará cuando la variable *activeBar* esté activada, que será el momento en el que la torre esté en fase *Stand.*

Podemos controlar la velocidad a la que la barra se llena mediante *barSpeed*, el tamaño de la barra mediante *distanceFactor* y los ángulos máximos de disparo con *maxAngle*.

De esta forma modificando estos valores podemos tener distintas barras si alguna torre lo necesitara.

Esta clase tiene tres etapas básicas. La primera es *Start* donde carga en memoria las imágenes que se necesitan para dibujar la flecha.

La segunda es *Update*, desde la cual comprueba si el atributo *activeBar* está activado, y en caso de estar activa se llama a *Aim*, donde controlará la rotación, potencia y disparo de la torre.

El método *Aim* obtendrá la posición del ratón y la posición del personaje sobre la pantalla mediante el uso de la cámara principal. Es decir que tendremos dos posiciones 2D de dos puntos sobre la pantalla.

Para calcular  cual es la rotación necesaria de la flecha para que apunte a dicho punto lo que se hace es conseguir el vector AB normalizado. Luego se realiza una restricción del vector por componentes para evitar tiros hacia atrás. A continuación se calcula la arcotangente con el vector normalizado. El método *Math.Atan2* devuelve el valor del ángulo en el plano cartesiano formado por el eje x y un vector que parte del origen (0,0) y termina en (x,y). Como devuelve el ángulo en radianes habrá que aplicar una transformación a grados centígrados. Finalmente se hace una restricción de sobre el ángulo máximo en sentido positivo y negativo y ya tenemos nuestra rotación.

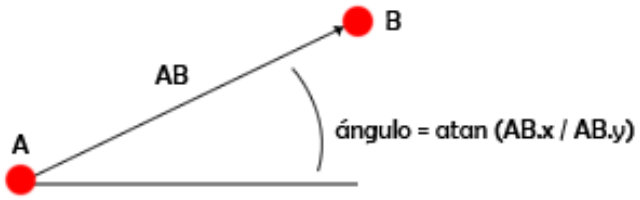



**Ilustración 47: Fórmula para calcular un ángulo entre dos puntos y el eje x**

Para calcular la fuerza se comprueba que se haya pulsado el botón fuera de otro elemento de interfaz, y en caso positivo se activa la variable *canPress*, mientras esta variable esté activa quiere decir que se ha pulsado el botón de forma correcta por lo que mientras se siga manteniendo pulsado se aumentará el *power* de la barra hasta que llegue al máximo, una vez se suelte el botón se disparará la flecha diciéndole a *character* que utilice el método *Shot* con el porcentaje de *power* y el vector dirección calculados. Por último se pone a false la variable *CanPress* al soltar el botón.

La tercera etapa es *OnGUI* donde se dibujarán las texturas rotando según lo calculado en el método anterior y dónde se aplicará el tamaño del poder según el porcentaje de carga. El método empieza comprobando que la barra esté activa para mostrarla o no. En caso afirmativo calculará el nuevo tamaño de la barra en función de la lejanía con la cámara, de forma que si acercamos la

cámara la flecha aumentará  pero si la alejamos disminuirá dando efecto de que la barra tiene un tamaño fijo como los personajes.

Una vez hecho esto se rotará el grupo de texturas utilizando *GUIUtility.RotateAroundPivot* pasándole el punto central y la rotación calculada. De esta forma ya tenemos la forma de apuntar lista y para el efecto de carga simplemente hacemos que la imagen cargada esté dentro de un contenedor y aumentamos o disminuimos su anchura en función del porcentaje de carga, de esa manera solo se mostrará una parte de la flecha cargada y dará la impresión de que se carga un contenedor vacio.

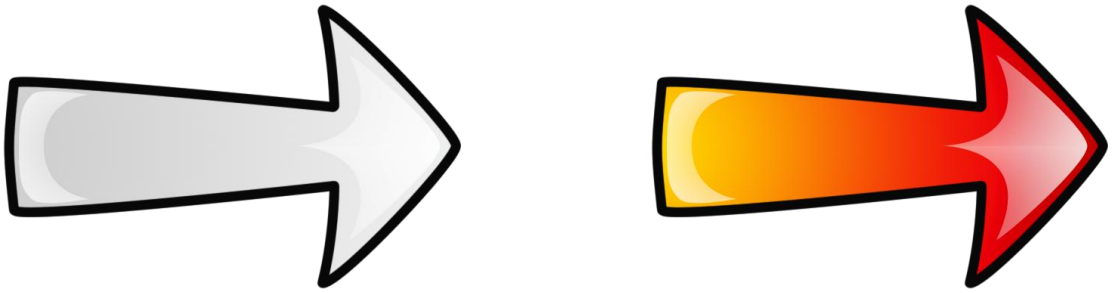


**Ilustración 48: Barra de energía**

#### 6.3.4.2. ButtonTower

Esta clase se encarga de representar la torre seleccionada así como su tiempo de recargo. La forma de funcionar de esta clase es muy parecida a la anterior clase.

Durante las etapas de carga se obtienen las imágenes que serán necesarias para representarse, tanto las que son iguales para todas las torres como los iconos representativos de la clase.

Luego en la etapa *Update* comprueba si la torre no está preparada, y en caso de no estarlo empieza a restarle a *timeRemaining* el tiempo transcurrido desde el anterior frame. Se calcula el porcentaje dividiéndolo entre el tiempo *rechargueTime*. Este valor será el tiempo que tarda en recargar una flecha desde que la dispara hasta que tiene otra en la mano. Sin embargo hay otro tiempo menor que sería el tiempo *chargueTime*, desde posición de reposo hasta que tiene una flecha en la mano. Estos dos tiempos habría que especificarlos para cada tipo de torre. Una vez que *timeTemaining* llega a cero la torre está preparada para otro disparo y se activa *isReady*.

Obviamente *timeRemaining* debe actualizarse cada vez que necesitemos recargar, es por ello que existen las funciones *StartRechargueTime* y *StartChargueTime* que son llamadas desde las subclases de *Character* cada vez que queremos que se inicie el contador.

El método RenderButton que es llamado desde OnGUI de GameCharacter utilizará los métodos de GUI para renderizar las diferentes texturas y formar los botones.

Para saber si el botón esta seleccionado o no y darle un color más oscuro se consultará el método *IsSelected* de *Character*. y para hacer el efecto de vaciado durante los *timers* se deberá bajar el grupo donde están contenido el fondo rojo de la imagen mientras que se hace más pequeño el *height* de dicho *group*, mientras que a su vez se tiene que ir moviendo hacia arriba la textura roja. Puesto que la textura roja está a su vez dentro del *group* estos movimientos opuestos se cancelan dando como resultado que la parte de arriba va desapareciendo al son que canta la variable *percent* calculada anteriormente.

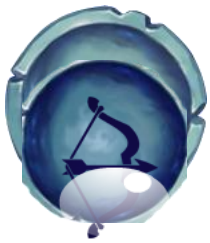


**Ilustración 49: Texturas del botón de selección de torre**

### 6.3.5. Efectos

Como ya hemos visto, las armas pueden asociarse con distintos elementos. Estos elementos se ven representados por *prefabs* que contienen sistemas de partículas con distintos efectos que han sido creados o modificados a partir de ejemplos descargados de la asset store de Unity.

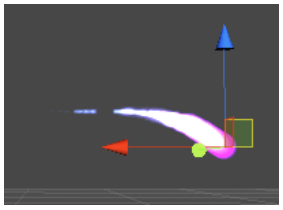


**Ilustración 50: Efecto por defecto**

El efecto por default es el utilizado por defecto puesto las flechas solas no se ven demasiado bien.

Este efecto está basado en un único sistemas de partículas con forma redondeada y que varían respecto de su movimiento por el mundo. Su color varia a lo largo de su ciclo de vida desapareciendo al final.

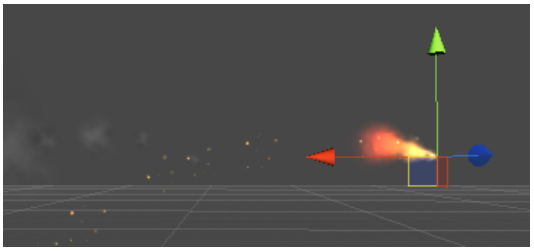


**Ilustración 51: Efecto de fuego**

El efecto de fuego está formado por tres sistemas de partículas. El primero es el fuego formado por formas especiales diseñadas para aparentar llamas, con un color que varia a lo largo de su ciclo de vida en tonos rojos. Los otros dos emisores crean partículas diminutas que simulan ser chispas y un humo que va dejando el rastro a medida que aparece. Este efecto también está ligado al mundo y su movimiento por él.

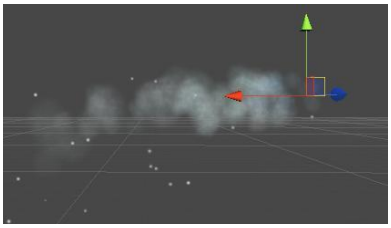


Ilustración 52: Efecto hielo

El efecto de hielo se basa en dos emisores. El primero genera una serie de partículas con forma de nube o vaho de hielo. Y el segundo genera unos copos de nieve que caen hasta el final. Ambos emisores están basado en el movimiento respecto al mundo pero el segundo contiene una gravedad mucho mas acrecentada para que caigan.

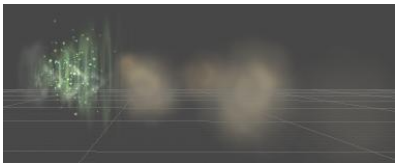


Ilustración 53: Efecto viento

El efecto de viento es de los más complejos que hay. Está formado por varios emisores, uno de ellos para el primer viento en la punta del proyectil, luego dos emisores mas para las curvas opuestas que van creciendo por el centro del tornado, un cuarto proyectil genera las partículas luminosas, el quinto genera los círculos de viento que forman el tornado y por ultimo uno más para generar el polvo que suelta. Estos emisores funcionan todos de forma local excepto el ultimo.

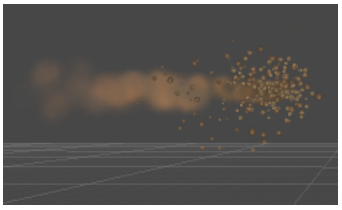


Ilustración 54: Efecto tierra

Este efecto está constituido por dos emisores. El primero funciona a ráfagas y genera las piedras que explotan mientras va avanzando el proyectil, el segundo son las nubes de tierra que suelta mientras explotan las piedras. Ambos emisores funcionan de forma local.

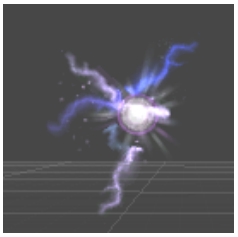


Ilustración 55: Efecto eléctrico

El efecto eléctrico está formado por varios emisores, el primero genera los círculos de luz que podemos ver en el centro, otro de ellos genera las luces en forma de rayos de luz que salen de la esfera central, otro genera las partículas de chispas que saltan desde el centro y por ultimo están los rayos de color azul que salen disparados. Todos estos efectos funcionan de forma local.

Ilustración 56: Efecto oscuro

El efecto de oscuridad contiene dos emisores, el primero es una bola de oscuridad que funciona de forma local y que se mantiene siempre en el proyectil y el segundo es un humo que se queda flotando en el aire, este si funciona de forma global y es afectado por el movimiento.

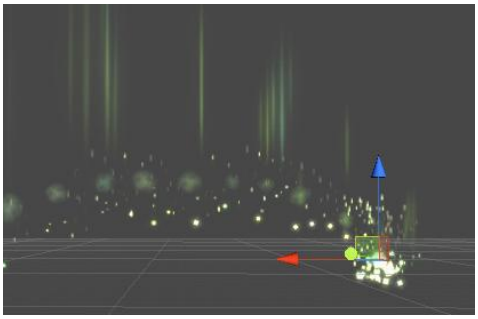


Ilustración 57: Efecto de luz

El último efecto es el de luz, está formado por cuatro emisores, el primero de ellos es el humo luminoso que se ve por el centro del recorrido, el segundo sería las luces más grandes que quedan por debajo, el tercerolas chispas que saltan por encima y por último los rayos de luz que caen desde el cielo. Todos estos emisores trabajan de forma global y son afectados por el movimiento.

## 6.4. Enemigos

Los enemigos están creados de forma que si queremos añadir un nuevo enemigo deberemos tener un modelo 3d, tanto con malla poligonal como estructura interna. Deberemos asignarle un cuerpo físico tanto al cuerpo como a la cabeza y añadirles algún *collider* añadiéndole los *tags enemy* y *head*. Además de esto también deberán tener un AudioSource así como un script de tipo Enemy y otro de tipo EnemyBar. Por último deberán tener asociado un AnimatorController con algún controlador que extienda de *Enemy.*

### 6.4.1. Modelo

Como hemos comentado anteriormente, los *prefabs* para un enemigo deben tener ciertas características físicas para que funcionen correctamente. En concreto necesitan de dos pares *RigidBody* y *Collider*. Uno para cada punto donde podemos golpearlo.

Necesitaremos buscar dentro de la estructura interna de huesos hasta encontrar con la cabeza y en ese momento asignar tanto cuerpo físico como un *collider* apropiado para la cabeza. Todos los disparos que acierten a la cabeza harán daño crítico. Pero para que esto ocurra debemos asignarle el tag *head*, de lo contrario no ocurrirá nada especial.

Para el cuerpo deberemos encontrar la primera transformada que se mueva con la animación pues si cogemos el *prefab* en sí, no conseguiremos que las flechas parezcan clavadas en el cuerpo del enemigo. En este punto deberemos de asignarle el tag enemy.

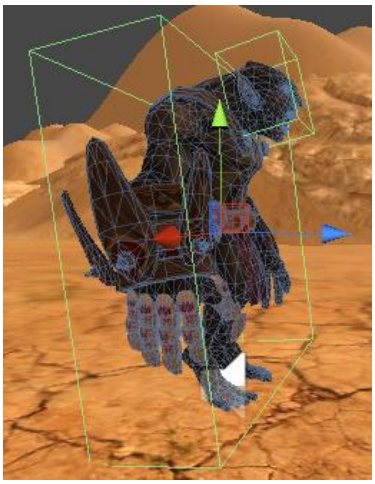


Ilustración 58: Box Collider de enemigos

Es también de vital importancia que el *prefab* entero esté contenido dentro del *layer enemy.* De esta forma evitaremos que los enemigos choquen entre ellos debido a distintas velocidades y que las flechas les puedan alcanzar. Esto es debido a la configuración del motor físico explicado anteriormente.

Las características de estos cuerpos físicos deben ser como las que aparecen en la imagen. Es decir no deben influir para nada en el movimiento de los personajes. Todo este movimiento estará controlado desde los AnimationController y la clase Enemy.

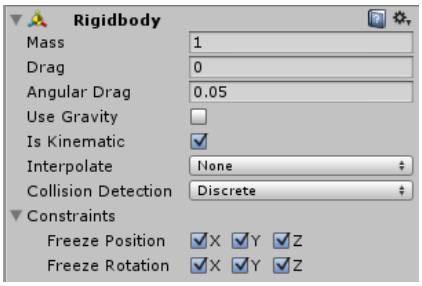



Ilustración 59: Rigidbody de un enemigo

Puesto que la mayoría de modelos obtenidos del asset store contienen animaciones básicas para caminar correr y recibir daño, es muy sencillo tener animaciones propias para cada modelo. Así que cada enemigo contiene su propia máquina de estados que hereda de la máquina de estados de Enemy.

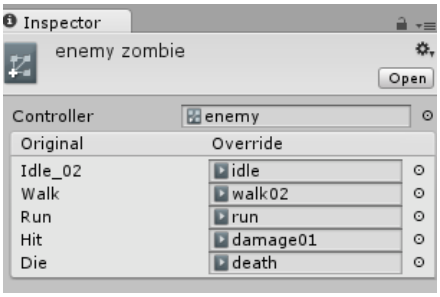



Ilustración 60: AnimatorController heredado

### 6.4.2. Scripts

Como hemos comentado hay dos scripts dentro de un enemigo: *Enemy* y *EnemyBar*.

La clase ***Enemy*** es la encargada de controlar el comportamiento de los enemigos. Esta clase contiene varios atributos clave para el enemigo como son su vida, su vida máxima, la velocidad, su resistencia al daño así como los sonidos que hará cuando sea golpeado y cuando muera. También tendrá relaciones con su animador, con la clase *EnemyBar*, con el *WaveController* y con *HashIDEnemy*.

En su etapa *Awake* iniciará todas las relaciones así como activará el atributo active de la máquina de estados. También calculará cual es la distancia desde la posición inicial hasta la cabeza y le pasará esta información a la barra de vida para que así sepa en qué lugar debe posicionarse.

Durante la etapa *Update* se comprobará si estamos en el estado *run* o *walk* de la máquina de estados. En caso de estar en el estado *walk* se procederá a trasladar al enemigo en la dirección forward, es decir hacia delante, en función de su velocidad y del *deltaTime*. En caso de estar corriendo se realizará lo mismo pero multiplicando la velocidad por dos.

A parte de estas dos etapas también tenemos el método *ReceiveDamage*, que es llamado por proyectil cuando detecta una colisión con un enemigo.

Este método en función del daño recibido restará vida al enemigo y mostrará un texto mostrando el daño recibido.

Lo primero es comprobar que la vida es superior a 0 pues de no ser así estaríamos recibiendo proyectiles sobre un enemigo muerto y no haría falta hacer nada. Una vez comprobado accionaremos el *trigger* para que el personaje realice una animación siendo golpeado. A continuación calcularemos el daño básico y el daño elemental a partir de los métodos de la clase *Damage*. Solo deberemos utilizar los métodos *calculateBasicDamage* partiendo del daño recibido y pasando nuestra resistencia y nos devolverá un valor numérico.

A la hora de mostrar el daño deberemos comprobar varios casos. El primero es comprobar si el daño es por barrera. De ser así debemos mostrar un texto apropiado *"Barrier damage"* *"Instant kill"*. Este daño se provoca cuando el enemigo llega a la base y choca con la barrera mágica. Lo segundo a comprobar es si el daño es crítico donde a parte del daño deberá mostrar el texto "*Critical*" y el tamaño de la fuente será más grande. Y por último tendríamos el caso normal donde se mostrarían ambos daños de forma normal. En todos los casos siempre se mostraría el daño básico con un color predeterminado y en caso de haber daño elemental se mostraría este debajo y separado por una sangría utilizando el color asociado al elemento.

Para mostrar estos textos para representar daño se hace uso del método *SpawnText* que permite instanciar un *DamageText* y modificar el texto, color y tamaño mediante parámetros. Luego lo posiciona encima del a cabeza del enemigo y a partir de ahí hará un desplazamiento y una desaparición, pero eso se explicará en el siguiente apartado cuando lleguemos a la clase *DamageText*.

Una vez mostrado el texto, se restarán los puntos básicos y elementales a la vida del monstruo y se modificará el porcentaje de vida de la *EnemyBar*. Además se hará sonar el clip de *hitAudio.*

Por último se comprobará si el enemigo debería morir por ese golpe. En caso afirmativo se actualizaría la máquina de estados para entrar en la animación de muerte, se avisaría al *WaveController* que ha muerto un enemigo, se haría sonar el *deathAudio* y se ejecutaría la corutina *Die*.

Esta corutina hace que el canal *alpha* de todos los materiales del personaje varíen de 0 a 1 durante varios intervalos. Esto se traduce en que el personaje parpadea volviéndose invisible mientras cae al suelo y al final desaparece del todo eliminándolo de la escena. Esto es necesario para no tener el escenario lleno de enemigos muertos por los suelos.

La clase ***EnemyBar*** se encarga de mostrar la vida de los enemigos en forma de barra que se descarga conforme recibe daño.

Esta clase contiene un porcentaje que será modificado por Enemy cada vez que recibe daño como hemos podido comprobar. Más tarde se representará la barra mediante texturas que serán dibujadas en la etapa OnGUI. La forma de mostrar esta barra sigue la misma estructura que la barra de poder vista en AimController, quitando el hecho de que no hace falta rotarla. Solo debemos colocarla justo encima de la cabeza de los enemigos y hacerle un escalado según la distancia con la cámara.

**Ilustración 61: Texturas de barra de vida enemiga**

Para mostrar el porcentaje de vida simplemente se multiplicará el *width* del grupo que contiene la textura roja por el atributo *percent*, haciendo que quede recortado el trozo de vida que ha perdido el enemigo.

## 6.5. Daño

Para manejar el daño se creó una clase ***Damage*** que contiene una serie de atributos que separan el daño en varias componentes. Además de esto hay

varios métodos para poder sumar y multiplicar daños o bonus así como métodos que calculan el daño recibido en función de una resistencia. De esta forma podemos dejar todo el tratamiento de daño a esta clase y abstraernos de cómo está estructurado el daño desde el resto de clases.

La clase *Damage* es la única que no hereda de *MonoBehaviour* puesto que no necesitamos ninguna etapa de ciclo de vida, solo lo utilizaremos como si fuera un tipo de dato.

Esta clase estructura el daño en las siguientes componentes:

- Daño básico
    - light
    - heavy
    - magic
- Daño elemental
    - fire
    - ice
    - elect
    - earth
    - wind
    - holy
    - dark

Hay varios constructores en función de si queremos crear un daño solo básico o con daño elemental también, dejando a 0 los valores no rellenados. También se han rescrito los operadores de suma para dos tipos *Damage* así como multiplicación entre dos tipo *Damage* y entre un tipo *Damage* y un *float*. Estos métodos suman o multiplican juntando los daños de cada tipo. Es decir *light* con *light, heavy* con *heavy*...

Para calcular el daño básico se multiplica cada daño básico por la resistencia a dicho daño concreto y se suma el resultado. De esta forma valores cercanos a 0 serían una resistencia alta a un tipo de daño y valores superiores a 1 serían resistencias bajas a un tipo de daño.

Para calcular el daño elemental se hace lo mismo pero con los distintos tipos de daño elemental.

Puesto que esta clase va a ser utilizado como un tipo de dato resulta muy útil declararla como serializable. Al añadir *[System.Serializable]* antes de la declaración de la clase conseguimos que la clase pueda ser representada en el editor mediante los atributos públicos que contenga. Esto es verdaderamente útil a la hora de probar diferentes combinaciones de configuraciones para balancear las armas y enemigos.

Para representar esta clase durante el juego se ha creado la clase ***DamageText***. Como ya hemos visto, esta clase se utiliza dentro de un *prefab* que contiene un *GUIText* y el propio script. Dicho *prefab* se instancia desde la clase *Enemy* cada vez que recibe daño y el script se ocupa de hacer una animación que lo hace desaparecer poco a poco mientras sube hacia arriba.

En la etapa *Start* esta clase personaliza el color y tamaño del texto así como transforma la posición en el mundo a una posición 2D en la cámara. Luego en la etapa *Update* mientras el valor *alpha* sea mayor que cero desplaza la posición del texto hacia arriba y disminuye el valor *alpha*. Cuando el valor llega a cero significa que texto ha desaparecido del todo y entonces se destruye el objeto de la escena.


**Ilustración 62: DamageText de un crítico**

## 6.6. Base

La base es el lugar donde intentan llegar las hordas enemigas, para proteger este lugar hay una barrera mágica protectora asociada a la vida que tenemos. Mientras tengamos vida la barrera protectora nos protegerá y eliminará a los enemigos que choquen con ella a costa de uno de nuestros puntos de vida.

De esta forma tenemos un *prefab* llamado *Life*, que es el controlador de la vida de la base y el que contiene el control sobre la barrera. Este objeto básicamente contiene un cuerpo físico estático y un *box collider* que detecte las colisiones de los enemigos. También contiene un sonido para cuando salta la barrera protectora así como un script asociado que controla el objeto.

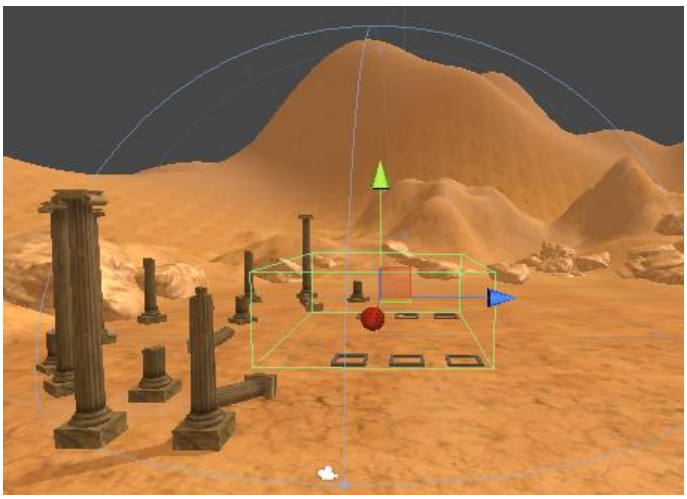


**Ilustración 63: BoxCollider de la base**

La clase ***LifeBar*** es la encargada de controlar la vida de la base. Esta contiene la vida actual que nos queda así como la vida máxima con la que se empieza la partida, también contiene referencias a los objetos que forman la representación visual de la vida así como referencia a *EndGameController* y a un efecto de partículas.

Esta clase representa visualmente la vida que nos queda de forma muy parecida a *EnemyBar*. Existe una textura de fondo sobre la cual se coloca otra textura que ira cortándose a medida que su contenedor se haga más pequeño según un porcentaje calculado dividiendo *life* entre *maxLife*. Por encima de esto se colocará un texto con el número de vidas que tenemos y un reflejo.

**Ilustración 64: Texturas utilizadas para representar la vida**

Durante la etapa *Start* se inician todas las relaciones necesarias así como *life* se inicia a *maxLife*. Más tarde se lanzan las ordenes de renderizado siguiendo la estructura explicada anteriormente.

Para controlar los contactos con los enemigos se ha desarrollado el método *OnTriggerEnter.* La diferencia con los eventos *OnCollisionEnter* es que no se produce una colisión física. Los objetos no saldrían disparados no su física cambiaría. Cuando ocurra este evento se comprobará que el objeto implicado sea de tipo enemy. En este caso se restará una vida y se recalculará el porcentaje. En caso de quedarnos con 0 vidas, el juego llegará a su fin habiendo perdido. De lo contrario se instanciará una explosión y se aplicará un daño infinito que hará morir al enemigo de forma instantánea. Una vez acabada la explosión se eliminará de la escena.

El efecto está formado por varios emisores de partículas. como podemos ver en la imagen tenemos dos círculos mágicos que aparecen así como una explosión de la cual se emiten chispas y unas partículas de rotura. Todas estas partículas cambian de color y medida a través de su ciclo de vida. El orden de aparición de estos emisores es el siguiente: primero aparece el círculo central, seguido de el segundo circulo que crece más rápido y en ese momento aparece la explosión central que hace salir disparadas el resto de partículas, como si se rompiera un cristal invisible.

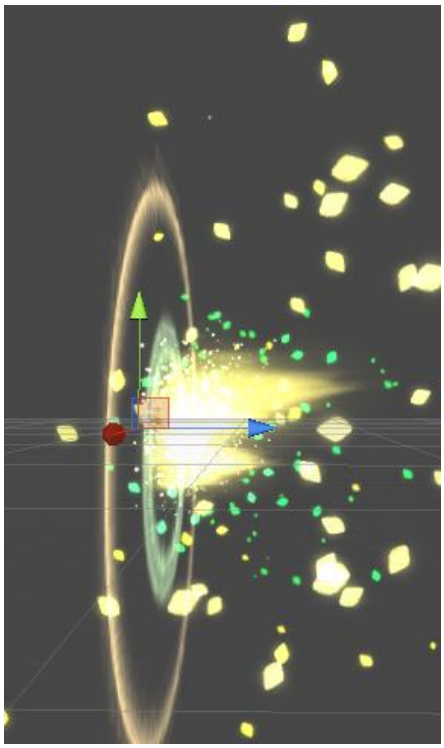


Ilustración 65: Efecto de ruptura de barrera

## 6.7. Hordas

Una horda es un conjunto de enemigos que aparecen en un orden y tiempo determinados en un lugar concreto de la partida. Estos objetos están representados por la clase ***Spawner***.

Estos objetos contienen varios atributos con los cuales se puede definir una horda desde el propio editor sin hacer falta saber cómo está programado.

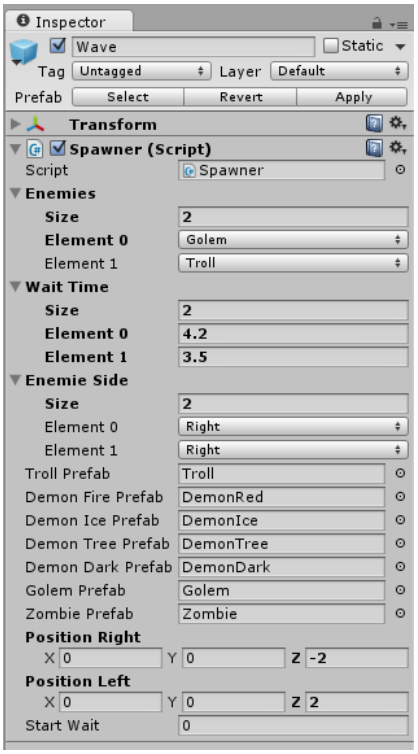



Ilustración 66: Inspector de un Spawner

Hay una lista con los tipos de enemigos que quieres colocar. Esta lista es un enumerado con cada uno de los tipos de enemigos existentes.

Existe otra lista con los tiempos de espera para que salga el siguiente personaje.

Y por último otra lista para marcar a qué lado se desea que aparezca el personaje. a la derecha o a la izquierda.

Para esto último es importante definir cuál es la posición de la línea derecha y cuál es la posición de la línea izquierda.

Otros atributos necesarios son el primer tiempo que se desea esperar hasta que salgan los personajes y las referencias a los *prefabs* de

enemigos. Estas relaciones ya están definidas en el *prefab* wave por lo que si instanciamos este *prefab* dentro de la escena y luego lo modificamos desde el editor solo tendremos que elegir que enemigos queremos poner, su tiempo de salida y el lado por el que saldrán.

El funcionamiento de este script es muy sencillo. Primero en su etapa *Start* obtienen una referencia al objeto *WaveContoller*. Y luego se quedan esperando a que el *WaveController* inicie la corutina *SpawnWave*.

Esta función básicamente instancia la lista de enemigos eligiendo el *prefab* adecuado según el enumerado *enemy*, luego en función del lado se instancia en una posición u otra y espera un tiempo determinado para instanciar el siguiente enemigo. Una vez los ha instanciado todos llama a la función *InitClock* de *WaveController* para que inicie la cuenta atrás antes de la siguiente horda y se autodestruye.

Si hubiéramos creado más enemigos y quisiéramos introducirlos en las hordas; deberíamos ir a la clase *Enums* y añadir los nombres de las clases en el enumerado, luego añadir atributos en la clase *Spawner* para poder tener una referencia al *prefab* del nuevo enemigo y por último relacionar el nombre del enemigo con el *prefab* dentro del *switch* del método *SpawnWave*.

## 6.8. Controlador de hordas

El controlador de hordas como su nombre indica controla cuando se lanza una nueva horda a escena. Mediante el uso de ***WaveController*** podremos mostrar en escena información sobre las hordas así como acelerar la entrada de estas.

Esta clase contiene atributos para todas las texturas así como variables para controlar los tiempos y referencias a todos los *Spawners* de la escena así como a *EndGamecontroller*.

Al igual que el resto de elementos de interfaz está compuesto por varias texturas. Para explicar la animación típica de reloj con cuenta atrás se ha realizado el siguiente esquema.

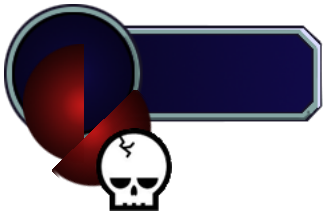


Ilustración 67: Texturas utilizadas para WaveController

Como podemos ver en la primera imagen, el control se basa en una base azul sobre la cual se coloca una primera mitad izquierda roja, y por encima de esta otra mitad derecha roja que va rotando respecto de su propio centro de origen.

De esta forma conseguiremos que se anime la primera mitad del circulo. Por encima de estos semicírculos colocaremos una calavera típica de enemigos.

Ilustración 68: Texturas para representar la segunda mitad del timer

Para la segunda mitad del *timer* cambiaremos el semicírculo derecho por un bloqueador con la forma del fondo. Y entonces rotaremos la parte izquierda del circulo. De esta forma la animación queda completa dando la apariencia de *timer* circular.

Esta animación se realiza durante la etapa OnGUI de forma similar a como hemos visto en todas las interfaces anteriores. Para realizar las rotaciones se utiliza un variable *percent* que contiene el porcentaje de tiempo que falta para la siguiente horda. Este porcentaje se calcula en el método *DoCountDown* que se llama desde *Update*. Simplemente se va restando *deltaTime* a una variable *timeRemaining* hasta que llega a 0.

Hay que tener en cuenta también que además de las texturas también se añade un texto indicando el numero de horda por el que estamos y el número máximo de ellas en el nivel actual.

El tiempo de espera entre hordas se actualiza a través de un método llamado *InitClock*. Este método comprueba que queden mas hordas por lanzar y en caso afirmativo edita la variable *timeRemaining* para que tenga el valor *startTime* así como pone la variable *isPaused* a false. Este método como ya hemos visto antes, es ejecutado por la clase *Spawner* cuando ha acabado de invocar a todos sus enemigos.

Ahora sabemos cómo se inicia el contador, pero nos falta saber cuándo se lanzan las hordas enemigas, pues como ya hemos visto para que una horda se lance hay que llamar al método *SpawnWave* del *Spawner*. Hay dos puntos clave donde se puede lanzar la siguiente horda, por lo que se ha creado un método llamado *NextWave* donde lanzamos la corutina y actualizamos el marcador. Este método es llamado desde el propio contador cuando su tiempo llega a cero y desde la etapa *Update* en caso que el usuario haga click en el botón.

El último detalle a comentar sobre esta clase, es que lleva un registro de todos los enemigos que tienen que ser eliminados. Es por esto que en la etapa *Start* hace un conteo de todos los enemigos de todas las hordas y se almacena dicho valor, y cada vez que un enemigo muere, este llama a *EnemyDead*. En este método se actualiza una variable con todas las muertes, ya sean por

barrera o por disparos, y si llega al número máximo de enemigos querrá decir que todos están muertos y hemos sobrevivido. Por lo cual llamaremos a *EndGameController* para ejecutar *WinGame*.

## 6.9. Escenas

Para la primera versión se han creado 3 escenarios jugables, con distintas hordas y enemigos. Para describir el proceso de creación de estos escenarios se hablará de ellos separando varios aspectos clave.

### 6.9.1. Música

Además de los efectos de sonido de enemigos, proyectiles, e interfaz, un juego siempre debe contener música de fondo. Así que se ha integrado para cada escenario su propio tema. Estos temas están sacados de la asset store y se pueden ver en la bibliografía.

Estos temas sonoros al contrario que el resto de sonidos no tienen un efecto 3d. Se ha decidido quitarlo pues el tema deberá sonar por igual en cualquier punto del escenario.

Para añadir el sonido se ha creado un *GameObject music* que contiene un *AudioSource* con el tema concreto así como un script *SoundObject*.

Es importante recalcar que al contrario que el resto de sonidos este tema sonará en bucle hasta que salgamos de la escena y sea eliminado. A no ser que el sonido este apagado claro.

### 6.9.2. Iluminación

Para la iluminación de los escenarios se han utilizado luces direccionales provenientes del sol en el cielo. De esta forma el efecto que produce es más realista y se consigue una iluminación básica.

A parte de esta iluminación direccional que tienen todos los escenarios, se han utilizado otros tipos de luces en ciertos puntos.

Por ejemplo en el siguiente escenario podemos ver como se ha utilizado una mezcla de luz puntual y halo para crear una luz que proviene de una esfera de energía.

**Ilustración 69: Iluminación puntual**

En la escena del castillo también se han utilizado luces puntuales para cada una de las hogueras que aparecían. Pero además de esto, se ha realizado un *bake* para procesar la iluminación de antemano y así aligerar la escena en tiempo real.

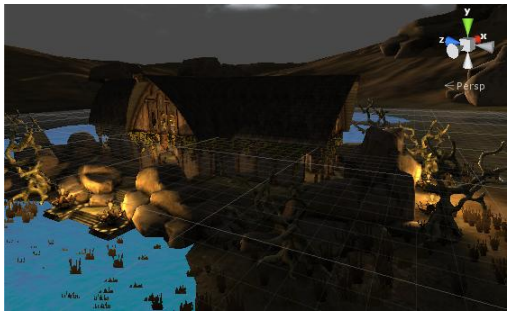



**Ilustración 70: Iluminaciones puntuales y bake**

### 6.9.3. Efectos

Para ciertos escenarios se han añadido varios efectos temporales, para dar a las escenas más dinamismo. Estos efectos son una nube de polvo en la escena del desierto y una lluvia en la escena de las montañas.

El efecto de la nube de polvo se ha conseguido mediante dos emisores de partículas, el primero emite una nube de polvo de un color más oscuro por la base del escenario y el segundo emite una cantidad de polvo mucho mayor que se dispersa por el aire.

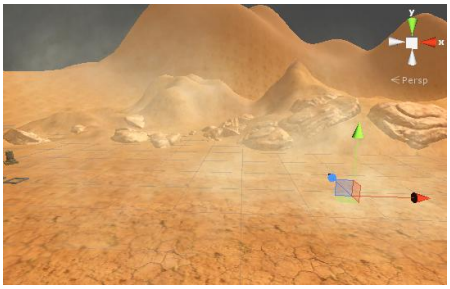



**Ilustración 71: Efecto de nube de polvo**

El efecto de lluvia se ha conseguido también a partir de dos emisores de partículas, el primero crea unas nubes de lluvia y el segundo crea las gotas de lluvia que caen desde esta nube.


**Ilustración 72: Efecto de lluvia**

### 6.9.4. Modelado

El modelado de los escenarios ha tenido puntos en común. Lo primero era buscar modelos que nos sirvieran como decoración. Un pack de piedras de montañas, un pack de piedras del desierto, ruinas, un palacio, texturas para la tierra...

Una vez estaban encontrados los elementos se creaba un terreno con el modelador de terrenos de Unity. Se le daba forma formando un camino llano y a su alrededor unas colinas o montañas haciendo que el terreno crezca mediante las herramientas que ofrece Unity. Luego se le daba color mediante las texturas intentando que se note el contraste entre el camino principal y el resto del escenario.

A continuación se instanciaban los diferentes modelos que se han encontrado y se rellenaba el escenario con ellos.

Por último se añadía un cielo mediante la edición del *skybox material*.

En el caso del escenario del castillo, el modelo del propio castillo ya venía con un terreno por lo que hubo que modificarlo a partir de él, tanto la escala como la creación del camino, recolocación de modelos...

Para el escenario del desierto se han utilizado modelos de rocas y ruinas de diferentes autores de la asset store y se han mezclado con un terreno creado desde cero.

Para el escenario de las montañas se han utilizado unos modelos de rocas de la asset store a los cuales se le han modificado los materiales para que adquieran ese tono verde brillante. Se han ido juntando diversos modelos de piedras para crear todo el escenario y las diferentes rocas que aparecen. El terreno y las montañas se han creado desde cero.

## 6.10. Fin de partida

El fin de partida se controla mediante los métodos *LoseGame* y *WinGame* que están en ***EndGameController***. Estos dos métodos se ejecutan desde *LifeBar* y *WaveController* respectivamente.

Al ejecutarse pausan el juego y activan una variable gameEnd que permite visualizar dos botones. Estos botones *restart* y *quit* están definidos en el método *OnGUI* y funcionan igual que sus análogos al pausar el juego.

Además de esto, se iniciará una corutina que realizará una animación de texto según hayamos ganado o perdido haciendo que aparezca el texto del centro de la pantalla haciéndose cada vez más grande.

# 7. RT-DESK

## 7.1. Introducción

RT-DESK es un núcleo de simulación de aplicaciones gráficas en tiempo real que sigue un paradigma discreto desacoplado. RT-DESK son las siglas de Real Time Discrete Event Simulation Kernel. Esto viene a ser un núcleo de gestión de eventos discretos ordenados temporalmente mediante paso de mensajes.

Uno de los principales objetivos de RT-DESK es el cambio de gestión de eventos continuos a discretos. Esto viene a ser dejar de muestrear a una frecuencia continua para pasar a declarar eventos discretos que se ejecutarán pasados x segundos. De esta forma todo el tiempo entre que pasa dicho periodo se puede reaprovechar para hacer otras cosas en CPU.

RT-DESK permite simular eventos continuos creando eventos a una frecuencia constante así como variar esta frecuencia según las necesidades del sistema. De esta forma en un mismo sistema pueden coexistir simultáneamente comportamientos discretos y comportamientos continuos, incluso, dentro del mismo objeto. Esta capacidad permite utilizar RT-DESK dentro de un motor de videojuegos de forma que aumentaría notablemente las capacidades iniciales del motor.

RT-DESK no es una aplicación y no funciona de forma aislada. Es un núcleo de simulación que hay que vincular con otro programa o motor gráfico. Una vez integrado este núcleo dentro del motor será capaz de encargarse de la gestión de eventos del sistema y de la comunicación entre objetos mediante paso de mensajes. RT-DESK no realizará ninguna tarea asociada al mensaje, simplemente se encarga de entregarlo y otro hará el trabajo por él. RT-DESK es una herramienta creada para dar soporte al desarrollo de aplicaciones gráficas en tiempo real mediante el paradigma discreto desacoplado.

## 7.2. Funcionamiento de RT-DESK

Como ya hemos comentado RT-DESK modela los eventos mediante paso de mensajes utilizando dos clases básicas: los objetos y los mensajes. Para enviar un mensaje el objeto debe formar parte de la clase *RTDeskEntity* y hacer uso del método *SendMsg* pudiendo elegir entre enviarlo de forma instantánea o utilizando un retardo. RT-DESK mantiene los eventos ordenados en función del tiempo en el que han de ser entregados para su ejecución. Una vez se pasa el tiempo de retardo se entrega dicho mensaje al *entity* asociado como destinatario y este deberá controlar el comportamiento ante este tipo de

mensajes. De esta forma RT-DESK controla los diferentes objetos de la aplicación mediante el paso de mensajes.

El encargado de realizar el paso de mensajes, recibiendo el mensaje y almacenándolo hasta su fecha de entrega es el *Dispatcher*. Este es el responsable de mantener el orden temporal de todos los eventos pendientes de envío. Cualquier objeto puede enviar un mensaje a otro objeto o a sí mismo, y todos estos mensajes pasaran por el *Dispatcher* antes de llegar a su destinatario. Si queremos que un objeto funcione a una frecuencia propia solo debemos controlar los retardos cuando se envíe mensajes a sí mismo. De esta forma podemos incluso controlar una frecuencia variable en función de otros factores. Simplemente con modificar el retardo del mensaje que se envía a si mismo, cada vez que recibe el mismo mensaje, sería suficiente. Es muy importante no sobrecargar al sistema cuando hagamos una simulación a frecuencia continua pues dependemos de la aplicación externa y debemos darle tiempo para que realice sus operaciones también.

Cada vez que el *Dispatcher* termina de enviar todos los eventos necesario finaliza su ejecución y devuelve el tiempo que ha de transcurrir hasta su próxima llamada. Es en este momento cuando se libera a la CPU de ciclos de ejecución innecesarios e improductivos.

## 7.3. Objetivos

Los objetivos básicos de esta integración es poder utilizar RT-DESK como centro de mensajería para los GameObjects utilizados por Unity.

La API de RT-DESK está escrita en C++ y para poder utilizar las clases desde Unity necesitamos transformar dicha API a un lenguaje que pueda ser interpretado por Unity. El lenguaje más parecido y la opción más clara es traducirlo a C# y hacer uso así de las diferentes clases que forman el API.

Aunque Unity permite utilizar DLL sobre código nativo en C o C++, para ello se necesitaría la versión PRO. Unity restringe el uso de DLL no solo a nivel de tipo de usuario y pagos, también restringe el uso de estas librería cuando la plataforma objetivo es aplicación web. Es por estos dos motivos que se ha decidido que era mejor obtener una traducción a C# en vez de integrar directamente el DLL resultante de la API.

Una vez traducida la API a C# se deberá encontrar una forma estructurada de utilizar las Entity de RTDESK desde scripts que hereden de MonoBehaviour. Puesto que Unity solo permite introducir scripts que hereden de dicha clase y que la herencia múltiple no está permitida por C# se ha realizado una estructura de clases para poder saltarnos este impedimento.

## 7.4. Traducción

Para la traducción del motor RT-DESK de C++ a C# se ha intentado conservar al máximo su definición original. Es por esto que se han conservado incluso los comentarios originales.

### 7.4.1. Clases traducidas

El resultado de esta traducción ha dado como resultado las siguientes clases en C#:

- HRTimer
- HRTimerManager
- RTDeskCEngine
- RTDeskDefCom
- RTDeskEntity
- RTDeskList
- RTDeskMsg
- RTDeskMsgDispatcher
- RTDeskMsgPool
- RTDeskMsgPoolManager
- RTDeskTime
- RTDeskTimers
- RTDeskTimer

Todas estas clases están contenidas dentro de ficheros .cs con su mismo nombre. Los nombres de las clases en el lenguaje original contenían el prefijo C de clase antes del nombre de la clase y después de RTDESK. La única clase que ha conservado este prefijo es *RTDeskCEngine* puesto que el *namespace* original es *RTDeskEngine* y esto producía problemas de nombre.

### 7.4.2. Defines

En C++ la palabra reservada define se puede utilizar con dos motivos básicos. Los defines que sirven para definir constantes y los defines que sirven para formar sentencias de precompilación.

El primer problema es fácil de solucionar, pues basta con cambiar esos defines por variables del mismo tipo pero añadiendo la palabra reservada const para definir que esa variable es constante. Por ejemplo:

```
public const int HRT_TIME_INVALID = -1;
```

Como podemos ver en el ejemplo todas las constantes que antes eran defines tienen todas sus letras en mayúsculas. Este es un detalle que hemos querido conservar para así ver más claramente que se trata de una constante a la cual no podremos cambiar su valor.

El segundo problema ha sido más difícil de solucionar. Algunos de estos defines se pueden utilizar en C#. Estos son los casos en los que se define un nombre que luego será utilizado dentro de un #if para saber si ejecutar un bloque de código u otro. El siguiente código es muestra de ello

```
#if RTDESKT_HRT
//[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void ResetClocks()
{
    SimulationClock = GetRealTime();
}
#else
//[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void ResetClocks()
{
    SimulationClock = SystemClock = clock();
}
#endif
```

Sin embargo otros de los define no podían utilizarse de esta forma. Se trata de los defines utilizados para definir el tipado de una variable o método en función del sistema objetivo de la compilación.

Esto con C# no se puede hacer así que se decidió utilizar el tipo necesario para la integración con Unity. En concreto este define era *RTTime*, que luego era traspasado a otros defines. Se utiliza para definir el tipo necesario para representar los *ticks* de tiempo en los que se basan los contadores. Para C# y la clase Time se utilizan enteros de 64 bits, es decir la clase *long* o lo que es lo mismo la clase *Int64*. Puesto que ambas clases representan lo mismo se decidió utilizar siempre la clase *Int64* para así saber que siempre que había un *Int64* en el código original estaba un define. Y cuando se utiliza *long* es que simplemente se utilizaba un *long* y no el *define*.

```
///The very next simulation time. It belongs to the first event in the time ordered buffer
Int64 NextMsgTime;

//if (NextMsgTime <= (SystemClock + SlackTime)), then simulate
//else stop simulation. Remain idle
RTDeskConfig    Configuration;

Int64 LastAlarm;///For accounting purpouses only
```

### 7.4.3. Punteros

Para la transformación de todos los punteros de C++ a C# simplemente se ha quitado el puntero y se ha cambiado el nombre de la variable quitando la p como prefijo que referenciaba que la variable era puntero. Puesto que en C# todas las clases son punteros, esto no ha resultado complicado. Solo había que quitar los punteros y asegurarnos que siempre que se utilizaba una clase se hacía uso del new para iniciarla antes.

Otros problemas con los punteros ha sido cuando se utilizaban como arrays en algunas variables de las clases *RTDeskMsgPool* y *RTDeskMsgPoolManager*. Para solucionar esto se ha cambiado el puntero por una variable de tipo array

con objetos de la clase en cuestión. Se han inicializado los arrays en el método constructor y de esta forma quedaba solucionado este problema.

#### 7.4.4. Destroy

Otro de los problemas a solucionar ha sido el uso de *destroy* desde el código original. En C# no podemos destruir por iniciativa propia un objeto de forma directa. Para eliminar un objeto debemos dejar la referencia a *null* y luego el *GarbageCollector* se encargará de eliminar esas variables por nosotros. Por lo que siempre que se utilizaba un *destroy* de algún puntero o variable se ha comentado para así evitar problemas al destruir variables.

#### 7.4.5. Inline

Siguiendo con otro problema de traducción nos hemos encontrado con la palabra reservada *inline*. Esta palabra hace que todo el código que aparece dentro del bloque del método sea incrustado cada vez que se llama a este método desde otro lugar. Esta directiva de precompilación se puede replicar en C# mediante el uso del encabezado *[MethodImpl(MethodImplOptions.AggressiveInlining)]* que está en el paquete *System.Runtime.CompilerServices.* El problema es que esta funcionalidad se ha añadido a partir de una versión de .NET que Unity por el momento no soporta, por lo cual ha quedado comentada. En caso de soportarse en futuras versiones solo habría que descomentarlas en cada uno de los métodos que lo utilizaba.

#### 7.4.6. SizeOf

Con la utilización de *sizeOf* hemos tenido también problemas pues en C++ se puede saber cuánto ocupará en memoria antes de compilar mediante esta operación, pero para C# esto solo sirve con clases básicas como int, long y otros tipos básicos. Para saber cuánto ocupa en memoria utilizamos *System.Runtime.InteropServices.Marshal.SizeOf* utilizando una instancia de la clase a utilizar.

Pero para que este método funcione es necesario utilizar una cabecera especial en la clase que queramos utilizar (*RTDeskMsg* y herederas).

```
[StructLayout(LayoutKind.Sequential)]
```

#### 7.4.7. PerformanceCounter

El último detalle a comentar sobre la traducción es la utilización de *QueryPerformanceCounter* y *FrequencyPerformanceCounter*. En C# se puede hacer uso de la dll "*Kernel32*.dll" y acceder a estos métodos internos. Esto ha quedado comentado pues para Unity, si se utiliza una importación de dll hace falta la versión PRO. De esta forma para la versión estándar se ha utilizado *DateTime*.*Now*.*Ticks* que aunque no tiene la misma precisión no hace falta importar nada. Y para la frecuencia se ha utilizado la clase *StopWatch* haciendo uso de la variable *frequency*.

## 7.5. Integración

Para lograr la integración se han creado cuatro clases más.

La primera clase a comentar es ***RTDeskMsgGamePlay***. Esta clase debería crearse de forma específica para cada juego. Esta clase contendrá las constantes de todos los tipos de mensajes que pensemos utilizar a lo largo del juego. Es una clase que solo sirve para saber el número de tipos que hay y sus nombres.

Luego por cada mensaje que queramos personalizar, deberemos crear una clase nueva que herede de *RTDeskMsg*. Por ejemplo si queremos que nuestro mensaje contenga ciertos atributos, como un float para almacenar un número, tendremos que crear una clase que herede de *RTDeskMsg* y añadirle dicho atributo.

La segunda clase a tener en cuenta es ***EngineInitiator***. Esta clase hereda de *MonoBehaviour* por lo que podrá asignarse a un objeto y a la escena. La clase contiene una instancia de *RTDeskCEngine*, la cual iniciará en la etapa *Awake*. Para iniciar el motor *RTDeskEngine* utilizará el método *StartUp*. Este método inicializa todos los objetos que hacen falta para arrancar el motor. Luego de esto se hará un *SetMaxMsgTypes* diciendo todos los mensajes que hay y en caso de tener mensajes que hayan sido sobrescritos, habrá que utilizar un *SetMsgType* por cada uno de ellos, para así poder iniciar de forma correcta el pool manager que está contenido dentro del motor.

Una vez todo está inicializado se ejecutará el método *Simulate* del motor RT-Desk, desde la etapa *Update* del *EngineInitiator*. De esta forma, esta clase será la única que sobrescriba *Update* de todos los *MonoBehaviour* que utilicemos.

La tercera clase a comentar es ***RTDeskMonoBehaviour***. Esta clase hereda de MonoBehaviour y será la clase padre de todos los objetos que queramos que estén dentro de RTDesk. Servirá de comunicación entre Unity y RTDesk.

Esta clase tendrá un método virtual que deberán implementar todas las clases que hereden de ella. Es el método *InitEntity*. Este método iniciará la subclase de *RTDeskEntity* que necesite. A parte de este método habrán dos atributos: uno que contenga una relación con el *RTDeskEntity* y otro que contenga la relación con el *EngineInitiator*.

Durante la etapa *Start* de esta clase, se obtendrá la relación con el *EngineInitiator* y se ejecutará el método *InitEntity* para iniciar la *RTDeskEntity* apropiada. Después de esto se enviará un mensaje de tipo *Start* a *entity* para que el objeto se active.

La cuarta clase necesaria para integrar en realidad son dos y forman un pack. Este pack es necesario para cada objeto que se quiera implementar utilizando *RTDesk*. Por ejemplo si queremos dotar a una esfera de un comportamiento

dentro del juego y queremos utilizar *RTDesk* deberemos crear una clase ***Esfera*** y otra clase ***EsferaRTDeskEntity***.

La clase *Esfera* heredará de *RTDeskMonoBehaviour* y como ya hemos explicado, sobrescribirá el método *InitEntity*. La clase *EsferaRTDeskEntity* heredará de *RTDeskEntity* y contendrá una referencia a *Esfera*.

De esta forma dentro de InitEntity haremos un new de EsferaRTDeskEntity guardando la relación en entity. Y luego le pasaremos nuestra referencia a dicha entity. Así ambos objetos tienen una referencia al otro.

La clase Esfera contendrá otros métodos que hagan uso del motor Unity, como pueden ser los métodos *setColor* y *move* que controlan los materiales y transformada del *GameObject* al que estén asociado. Estos métodos serán utilizados desde *EsferaRTDeskEntity*.

Por su parte *EsferaRTDeskEntity* sobrescribirá el método *ReceiveMessage* para controlar los mensajes que recibe. En función del tipo de mensaje que haya recibido llamará a una función u otra de esfera. Por ejemplo al recibir el mensaje de tipo *start*. *EsferaRTDeskEntity* iniciaría todas las variables que necesitará y ejecutaría los métodos de esfera que viera oportunos.

Como pequeño resumen diremos que para introducir cualquier objeto dentro del *RTDesk* y que los mensajes estén controlados por este motor solo debemos seguir el mismo patrón. Creamos la clase que herede de *RTDeskMonoBehaviour* y añadimos todo el comportamiento que añadiríamos si la clase fuese a ser controlada por unity como de costumbre. Creamos una clase que herede de *RTDeskEntity* y que contenga una relación con la anterior clase. Sobrescribimos el método *ReceiveMessage* y enlazamos los tipos de mensajes con los métodos creados

# 8. Conocimientos aplicados durante la carrera

Para desarrollar un videojuego hacen falta conocimientos muy dispersos de varias ramas de la informática. No solo hace falta saber programar en un lenguaje, hacen falta conocimientos matemáticos, de física, persistencia de datos, diseño de interfaces, de niveles, conocimientos sobre usabilidad, gestión de proyectos, inteligencia artificial, posicionamiento en un mundo virtual, conocimientos gráficos, etc, etc... Y todo esto sin salirnos del trabajo del informático, hacen falta gente de múltiples campos, por lo que también viene bien saber lo básico sobre los otros campos para ser capaces de coordinarse correctamente y entender el coste de ciertas tareas.

Para relacionar todo estos conocimientos con los aprendidos durante la carrera se va a hacer una relación entre tipos de conocimientos y asignaturas. De esta forma esperamos que esto sirva para dar más peso a ciertos aspectos de la carrera en un futuro.

- **Programación**: Una de las principales tareas en la creación de un videojuego es la programación. Para este proyecto se ha utilizado C#. Lenguaje con el cual se han realizado las prácticas de varias asignaturas. Asignaturas como PRG en la que vimos Java y la orientación a objetos han ayudado. Este campo está más que cubierto a lo largo de toda la carrera con todas las asignaturas de programación que hemos tenido.
- **Matemáticas**: Para esta parte han sido de bastante ayuda asignaturas como *Análisis matemático* y *Matemática discreta y algebra* que se dieron al principio de la carrera. Una buena base matemática es esencial para cualquier desarrollo de algoritmos. Y en un videojuego siempre hacen falta. Han sido de especial ayuda asignaturas de especialización (*Diseño asistido por computador, fabricación asistida por computador, Gráficos por computador*) donde ,aunque no se explicaba de forma explícita, se ha trabajado mucho con matemática matricial
- **Física**: Para el desarrollo de la parte física de este juego se ha utilizado el motor físico de Unity pero para ello hace falta entender una serie de conocimientos físicos básicos. Estos conocimientos se han aprendido antes de llegar al a carrera pues la física que se ve durante la carrera es una física electrónica, y todo lo que te pueda servir para las simulaciones físicas vienen de antes. No hubiera estado mal tener alguna asignatura más específica para simulación de entornos físicos reales.
- **Máquinas** de estado: El conocimiento sobre máquinas de estado resulta muy útil a la hora de desarrollar el comportamiento de los personajes así como asociar sus animaciones a estos estados. Estos

conocimientos se han visto reforzados en asignaturas como *Procesadores de lenguaje o Inteligencia artificial.*

- **Persistencia** de datos: La persistencia de datos es algo esencial en cualquier desarrollo software y eso se ve reflejado en las asignaturas. Hemos dado persistencia de datos en muchas asignaturas y en muchas formas, desde ficheros XML hasta bases de datos. Esto se puede ver en asignaturas como *Bases de datos, Arquitectura de bases de datos, Ingeniería de la programación...*
- **Usabilidad**: Una de las partes más importantes de un software es que sea usable e intuitivo, y para eso hay varias asignaturas optativas que han ayudado bastante. Algunas de ellas son *Integración multimedia o Desarrollo de aplicaciones en entornos web.* Esa última aunque no está enfocada a la usabilidad, el profesor nos enseño bastante sobre como estructurar bien una interfaz.
- **Gestión de proyectos y metodologías**: Un videojuego es como cualquier otro desarrollo software. La gestión de proyectos es esencial, sobretodo cuando el proyecto es más grande. Para estos conocimientos se pueden encontrar asignaturas en la especialidad de desarrollo software. *El proceso del software, laboratorio de desarrollo de sistemas de información, Ingeniería de requisitos...*
- **Gráficos**: Un videojuego actual sin conocimientos sobre gráficos 3D y otras características típicas de estos entornos es impensable. Para ello han sido especialmente útiles las asignaturas de la especialidad de industriales basadas en este campo: *Gráficos por computador, tratamiento de imagen digital, diseño asistido por computador, producción de imagen digital...*
- **Sonido**: Al igual que con los gráficos, el sonido es indispensable. Hay muy pocas asignaturas que toquen el sonido, pero hay una asignatura optativa que me enseño lo básico para poder modificar cosas sobre ciertos efectos de sonido sin necesitar a un experto para poder hacerlo. Esta asignatura fue *Introducción a la síntesis edición y postproducción de audio*.
- **Control de versiones**: Hoy en día es necesario contar con experiencia utilizando controladores de versiones. El trabajo siempre será en grupo y es importante tener esto en cuenta. En *laboratorio de sistemas de información* se trabajo con un controlador de versiones, pero esto fue decisión del grupo de trabajo más que de la asignatura. Se debería tener más en cuenta este factor e integrarlo en todos los proyectos que se pueda.

# 9. Conclusiones

Este proyecto ha sido complejo y me ha costado más de lo que pensaba. Pero ha merecido la pena el esfuerzo por varios motivos:

- Primero de todo, he conseguido romper mano con el desarrollo de videojuegos en entornos 3D. Es importante haber empezado este primer intento, pues ahora será más fácil intentar acceder a algún puesto de trabajo relacionado.
- He aprendido que este trabajo es difícil, y que necesita de mucha gente. Sobre todo de gente experimentada en las primeras etapas de análisis y diseño. Tener a alguien con experiencia que te guie durante el desarrollo habría sido de mucha ayuda. Pero de haberlo tenido no me habría dado cuenta de la falta que hace.
- He aprendido sobre el entorno Unity con C#, y gracias a la traducción he aprendido muchos detalles que desconocía sobre C++. Y estos dos lenguajes y el entorno Unity son unas de las habilidades más pedidas para este tipo de desarrollos, por lo que me ha venido muy bien esta experiencia.
- He cogido experiencia como desarrollador independiente, buscando yo todos los recursos y aprendiendo a completar características con los pocos recursos de los que disponía. De esto he aprendido que para ser desarrollador independiente hace falta más experiencia y para embarcarse en proyectos profesionales más todavía.

Por todo esto creo que el proyecto ha merecido la pena y que todo el esfuerzo ha tenido su recompensa. Pues he cogida mucha experiencia en el campo, que era de lo que se trataba.

# 10. Bibliografía

En este punto se detallará la bibliografía utilizado tanto como aprendizaje como para recursos:

Libros

- Unity GameDevelopment Essentials by Will Goldstone
- Unity 3D Game Development by Example: Beginner's Guide by Ryan Henson Creighton
- Unity 3 GameDevelopment: HOTSHOT by Jate Wittayabundit

Links

- unity3d.com
- forum.unity3d.com
- unityspain.com
- stackoverflow.com
- cgcookie.com
- tf3dm.com
- turbosquid.com
- autodesk.com
- sonidosmp3gratis.com

Asset Store

- ambient sample pack v1.0 by electrodynamics
- Ancient ruins in the desert by nekcom entertainment
- Elementals v1.1.1 by G.E.TeamDev
- Fantasy knight v1.01 by Bunt Games
- Free Game Music by Vertex Studio
- Free Rocks by TripleBrick
- Goblin ranger by Shunsuke Yamamoto
- Maze Element Demon by Andres Olivella
- Maze Element Ice Golem by Andres Olivella
- Palace of Orinthalian vFinal by geartechgames
- Sandstone Desert Rocks by Funky Llama
- Shanty Town by Unity Technologies
- Sky FX Pack by Unity Technologies
- The earthborn Troll by Sou Chen Ki
- Tyrant Zombie by M.eye

# 11. Anexo A: GDD del juego


## Design Document for:

## Beast's Retreat

**The Ultimate Tower Defense Game**



Version # 1.00

# Tabla de contenidos

# Visión General del juego

## Filosofía

**Punto filosófico #1**
Plantear retos a los usuarios que les atraigan los juegos de estrategia/habilidad y poner a prueba su capacidad para superar complicados niveles.

**Punto filosófico #2**
Entretener al usuario y aislarlo de la realidad y los problemas que conlleva.

**Punto filosófico #3**
Despertar interés en el usuario por conseguir todos los elementos y descubrir todas las combinaciones posibles.

## Preguntas frecuentes

**¿Qué es el juego?**
Consiste en defender tu base de los diferentes enemigos que irán apareciendo, mediante el uso de de diferentes torretas que el jugador podrá manejar. Utilizando estas torres deberemos apuntar y disparar a los enemigos para acabar con ellos y evitar que destrocen la base.

**¿Por qué se ha creado el juego?**
Este tipo de juegos ha dado resultados en el pasado, pero actualmente no hay ningún juego sobresaliente en este ámbito. Si se gestiona bien podría llegar a ser un juego adictivo que guste a un amplio sector del mercado. Juntando esto con la capacidad para expandir el juego mediante nuevas torres y elementos para mezclar podemos crear un efecto adictivo en el usuario para conseguir todos los elementos y combinaciones.

**¿Dónde toma lugar el juego?**
Transcurre en un mundo de fantasía que combina diferentes elementos medievales, mitológicos y heroicos. Este tierra se llama X y contiene paisajes tan variados como llanuras con castillos, bosques, desiertos, montañas de esmeralda, lagos y ciudades costeras.

**¿Qué puedo controlar?**
El control del juego está basado en las diferentes torres que podemos utilizar. En el juego tendremos una base con 6 torres distintas, las cuales utilizaremos para disparar y acabar con los enemigos.

Estas torres deberán crearse a partir de diferentes elementos que el usuario podrá combinar para obtener nuevos objetos que posteriormente se utilizarán en los mapas.

Podemos elegir que torres utilizar para cada misión, siendo de vital importancia elegir las torres adecuadas para mejorar la eficacia de los disparos según los tipos de enemigos que aparecerán.

**¿Cuántos personajes puedo controlar?**
El jugador podrá manejar diferentes tipos de torretas dependiendo de las combinaciones que haya realizado y el número de éstas que permita el nivel en el que se encuentre. Independientemente del número de torretas disponibles, solo podrá controlarse una a la vez.

**¿Cuál es el objetivo?**
El objetivo final es defender la base de las hordas enemigas y lograr eliminar a todos los enemigos sin que nuestra vida llegue a cero.

**¿Qué tiene diferente?**
Este juego se diferencia del resto por combinar elementos de varios géneros, como es la elaboración de torretas a partir de elementos que podremos obtener de diferentes formas.

# Características

## Características generales

Modelos 3D

Vista 2D.

Cambio de cámara

32-bit color

Diferentes niveles de tamaño pequeño.

## Editor

Fácil uso.

Integrado en el juego.

Crea nuevas torretas mediante combinación de elementos.

## Gameplay

Jugabilidad 2D.

Selección de trayectorias de proyectiles en base a altura y potencia.

Modo de juego survival por oleadas.

# El mundo del juego

## Visión general

El mundo en el que se sitúa el juego será de fantasía con influencias medievales, mágicas y referencias a otros juegos.

Este mundo contendrá escenarios y lugares muy diversos como: praderas verdes en colinas, bosques élficos, cuevas infestadas de murciélagos , minas enanas, poblados humanos, llanuras de orcos, desiertos, tierras volcánicas, montañas de esmeralda...

## Características del mundo

Los diferentes escenarios tendrán efectos especiales según el clima de la zona. Esto puede variar desde una niebla que cubre unas laderas, a lluvia o incluso tormentas de arena.

## El mundo físico

### Visión general

El mundo físico estará muy limitado, de forma que solo podremos ver el escenario en el cual estemos actualmente. Solo influirán en la jugabilidad la pendiente o silueta del escenario y el viento. De forma que el mundo simplemente servirá de ambientación.

### Lugares clave

Las diferentes localizaciones clave serán los distintos niveles del cuál conste el juego. Habrán una serie de niveles los cuales deberemos ir superando para seguir al siguiente. El orden de estos niveles será crucial pues si no conseguimos superar el primero no podremos acceder al segundo y así sucesivamente.

### Viajes

Los viajes por el mundo se realizarán a través de un mapa de selección de nivel. En este mapa podremos ver todo el mundo creado y seleccionar el nivel que queremos jugar y por tanto viajar a través del mundo. Como ya se ha comentado para poder ir a los escenarios más lejanos deberemos superar primero los niveles que nos llevan a estos.

### Escala

La escala a la que se representará el mundo será siempre la misma. Obviamente será una escala reducida para poder observar el escenario con perspectiva y así atacar al enemigo siguiendo una estrategia.

## Sistema de renderizado

### Visión general

El juego se renderizará utilizando una perspectiva horizontal situando la cámara en el eje z y apuntando a x e y. Será una vista típica 2D utilizando un sistema de renderizado 3D anclado en dichos ejes.

### Renderizado 2D/3D

Se utilizará el motor gráfico que ofrece Unity. De esta forma al utilizar un modelo 3D luego podremos realizar diferentes movimientos alrededor del mundo para cambiar entre diferentes vistas.

## Cámara

### Visión General

Básicamente podremos variar entre dos puntos de cámara distintos (z y -z). Este movimiento de cámara permitirá situarnos a ambos lados de un camino pudiendo así tener más torres a elegir.

### Detalle de cámara #1

Movimiento en el eje x. Podremos movernos alrededor del eje x de una forma limitada. Es decir podremos movernos desde el punto de inicio de los enemigos hasta nuestra base.

Movimiento en el eje y. Podremos movernos alrededor del eje y de una forma limitada. Es decir podremos movernos desde el suelo hasta la altura de la torre más alta.

Movimiento en el eje z. Podremos movernos alrededor del eje z de una forma limitada. Es decir podremos movernos desde el punto más cercano a la fila de torres hasta un lugar más alejado que nos permita ver el máximo rango posible.

### Detalle de cámara #2

El segundo tipo de movimiento nos permitirá realizar una translación en la cámara para intercambiar la posición z de esta. Es decir que solo podrá estar en dos posibles posiciones z opuestas y su punto de mira siempre será el mismo. Será como mirar una calle desde un lado o desde el otro. Esto nos

permitirá tener dos filas de torres disponibles, pero solo podremos utilizar una de las filas según en qué posición estemos.

## Motor del juego

### Visión general

El motor físico del juego tendrá que tener en cuenta varias cosas en mente. Deberá controlar a los enemigos, los proyectiles y limites del escenario.

### Detalles del motor

Se deberá poner especial cuidado en las trayectorias de los proyectiles. Estos proyectiles dependiendo de sus características físicas realizarán diferentes trayectorias para una misma altura y fuerza. El peso y la forma de los proyectiles afectará a la trayectoria haciendo que no solo importe la altura y la potencia con la que se lancen.

### Detección de colisiones

Será de vital importancia hacer un buen uso de las colisiones de los proyectiles con los enemigos. Para asegurarnos de no perdernos ninguna colisión se hará un seguimiento especial de los proyectiles teniendo en cuanta que van a llevar una velocidad importante y que se debe comprobar adecuadamente su colisión.

También se deberán comprobar las colisiones de los enemigos con el límite de nuestra base y así eliminar su aparición en el mundo y evitar sobrecargas así como hacer lo mismo con el lugar de inicio de los enemigos y los proyectiles.

## Modelos de iluminación

### Visión general

El modelo de iluminación que utilizaremos será un modelo ambiental que dependerá de cada escenario. Se utilizará una iluminación basada en luces direccionales basándonos en la posición del sol en cada uno de nuestros escenarios. A parte de estas luces también deberemos tener en cuenta ciertas luces puntuales que emiten algunas partes de los escenarios como podrían ser hogueras o esferas de energía.

# Disposición del mundo

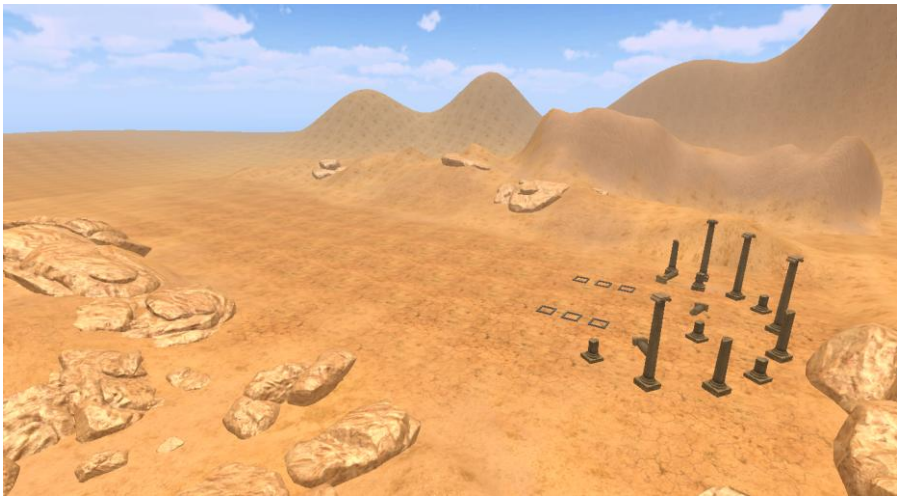
## Visión general

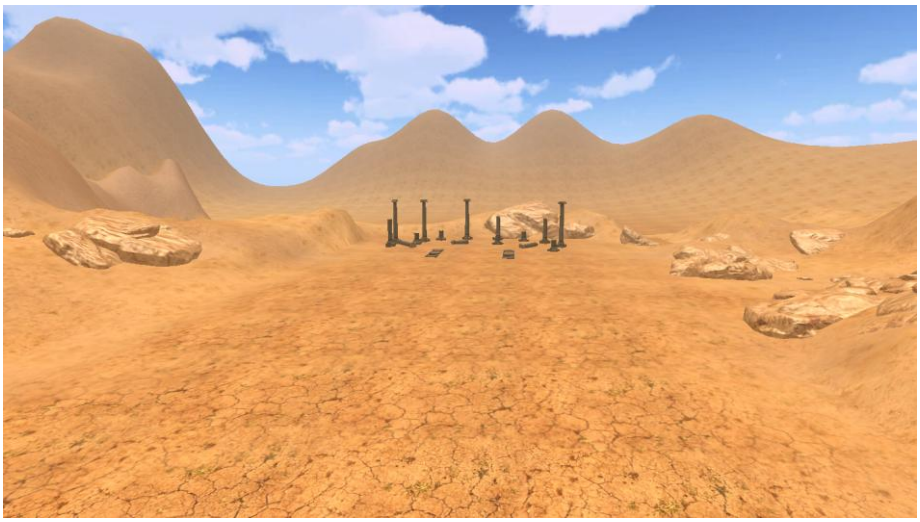
Los diferentes escenarios serán explorables a través de un mapa donde podremos seleccionar el nivel. A continuación se mostrará un mapa del mundo así como diferentes escenarios de este.

## Detalles del mundo #1

Mapa del mundo

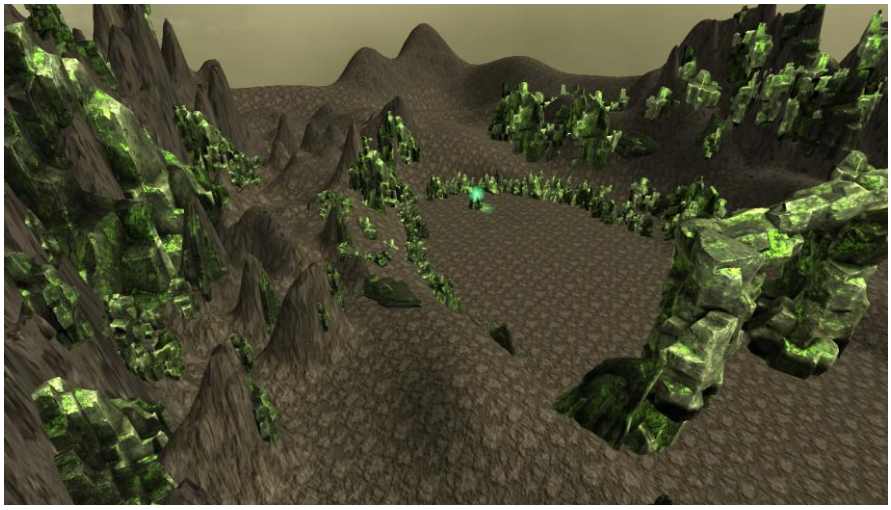
**Detalles del mundo #2**
Desierto de Janna

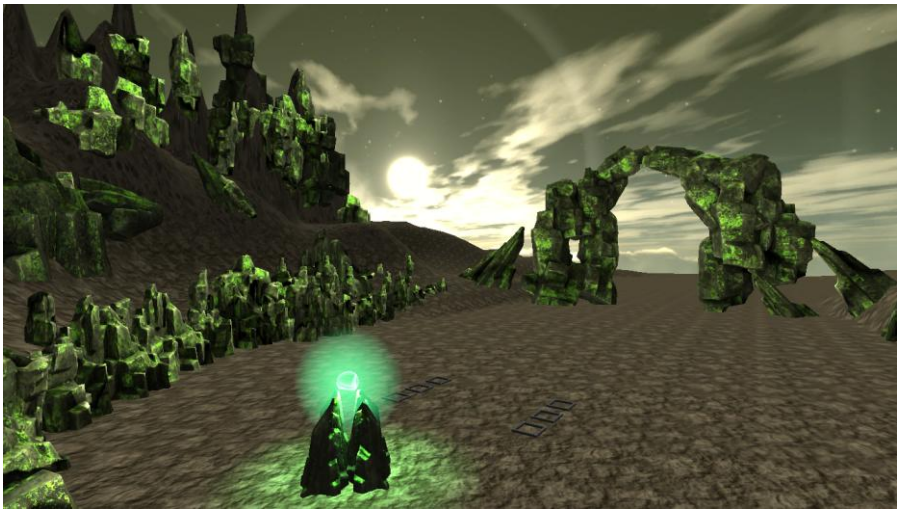


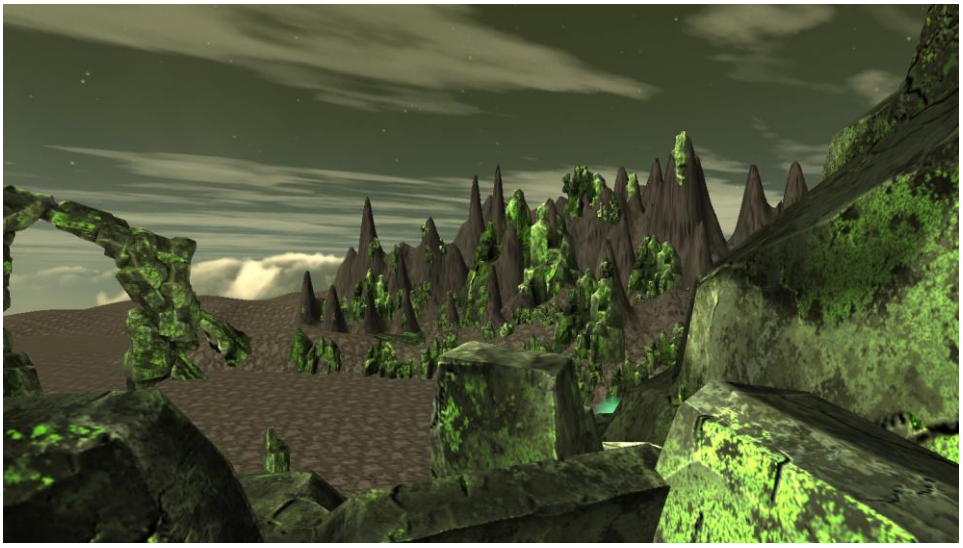


## Detalles del mundo #3

Emerald Mountains

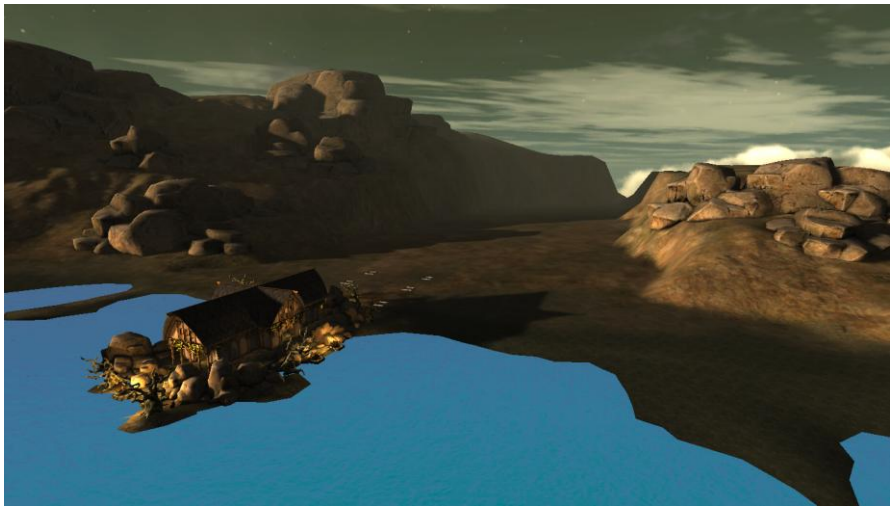


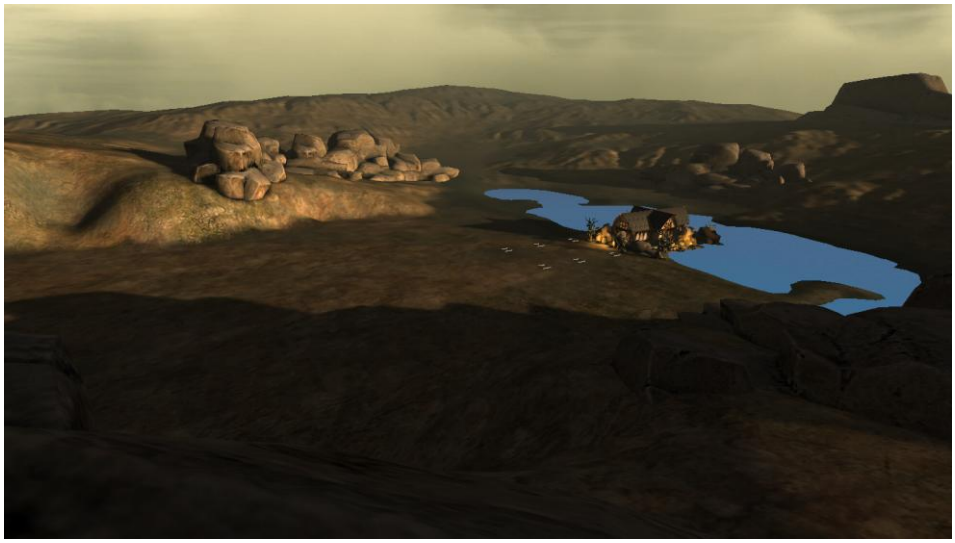




## Detalles del mundo #4

Palace of Orinthalian





# Personajes del juego

## Visión general

El personaje principal en este juego será un estratega. Es decir, nosotros mismos al elegir los distintos tipos de torres a utilizar en cada mapa. También seremos capaces de crear distintas torres y artefactos que utilizar luego.

Nuestro personaje carece de relevancia en el juego pues se enfoca más a las diferentes torres que trataremos como armas y a los personajes enemigos a abatir.

## Enemigos y monstruos

Los enemigos o monstruos serán variados y de diferentes clases. Tendremos diferentes clases de monstruos dependiendo de su raza y tipo.

Los diferentes tipos de enemigos serán los siguientes:

-Básico: este será el enemigo básico, sin más resistencia que el resto ni habilidades especiales. Sus características dependerán únicamente de su raza.

-Ágil: este será un enemigo más rápido que el resto y para compensar tendrá una resistencia reducida respecto al tipo básico.

-Tanque: este enemigo será un poco más grande que el resto y con una resistencia superior a la media. Tendrá más debilidad a los daños por magia que a los físicos. También será más lento que el resto de unidades.

-Mago: este enemigo tendrá una vida y resistencia básica excepto para los daños especiales, a los cuales será más resistente. También podrá utilizar hechizos varios dependiendo de su raza para mejorar a sus compañeros.

-Jefe: este enemigo tendrá una resistencia superior a la media y podrá inspirar a sus compañeros haciendo que mejore su vida y velocidad. También hará que se generen enemigos extra mientras esté en el campo de batalla.

Las distintas razas de los enemigos serán las siguientes y pueden ofrecer distintas habilidades.

-Bárbaros: tendrán una vida y una resistencia media para todos los daños. La habilidad de sus magos curará a las criaturas cercanas.

-Orcos: tendrán una vida y resistencia superior. Tendrán una resistencia un poco más reducida en daño ligero que en pesado. La habilidad de sus magos aumentará todas sus resistencias durante un tiempo limitado.

-Elfos: tendrán una vida muy superior a la media pero una resistencia normal. Tienen una ligera posibilidad de esquivar daños físicos. La habilidad de sus magos los volverán invisibles durante un tiempo limitado(Pero se les puede dañar igualmente).

-Bestias aladas: tendrán una vida y resistencia menor a la media. Tienen la habilidad de volar lo que evita posibles daños por explosiones. La habilidad de sus magos los hacen más resistentes a las magias.

-Híbridos humanos/bestia: tendrán una resistencia superior a la media y serán más rápidos. Cuando su vida sea baja correrán aún más rápido. La habilidad de sus magos hace que su vida se regenere durante un tiempo limitado.

-Enanos: tendrán una resistencia superior a la media pero serán más lento. Cuando sean dañados por primera vez se volverán inmunes a todo daño y excavaran para avanzar por debajo del escenario durante un tiempo limitado. La habilidad de sus magos los hacen más resistentes a la magia.

-Muertos: tendrán una vida menor a la media. Cuando mueren tienen una posibilidad de volver a la vida de nuevo. La habilidad de sus magos hace que todos los muertos cercanos resuciten, sean muertos o no.

-Elementales: tendrán una vida normal y unas resistencias más extremas a los elementos basados en su tipo de elemento. La habilidad de sus magos hará que sean totalmente inmunes al daño elemental en el que estén basados.

# Interfaz de usuario

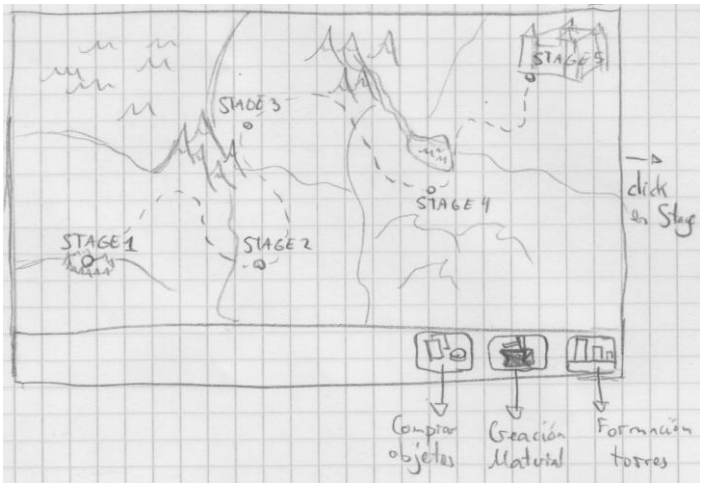
## Visión general

A continuación se muestran diferentes bocetos de la interfaz de usuario y las funcionalidades aportadas por el juego:

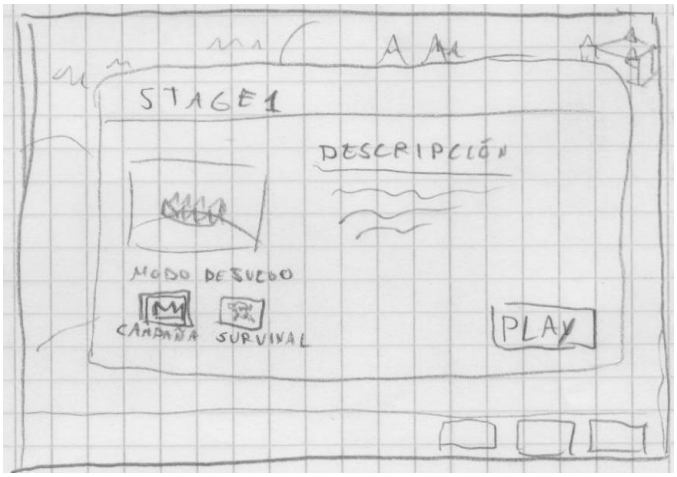
**Detalle de interfaz de usuario #1 – Pantalla de inicio**


**Detalle de interfaz de usuario #2 – Seleccionar partida**

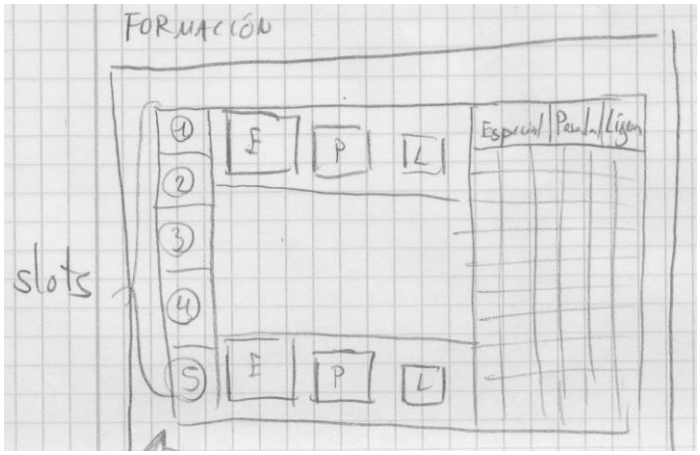

**Detalle de interfaz de usuario #3 – Seleccionar escenario**



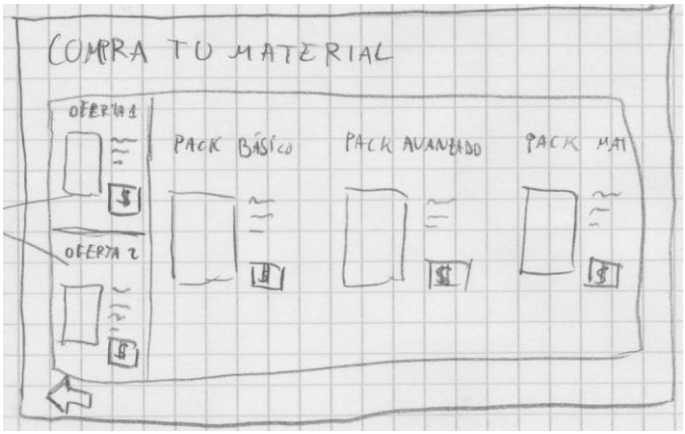
### Detalle de interfaz de usuario #4 – Descripción de escenario y selección de modo



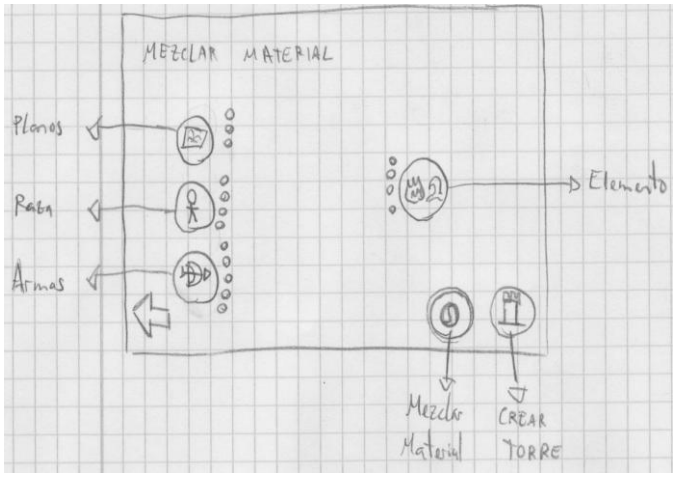

### Detalle de interfaz de usuario #5 – Selección de la formación de defensa



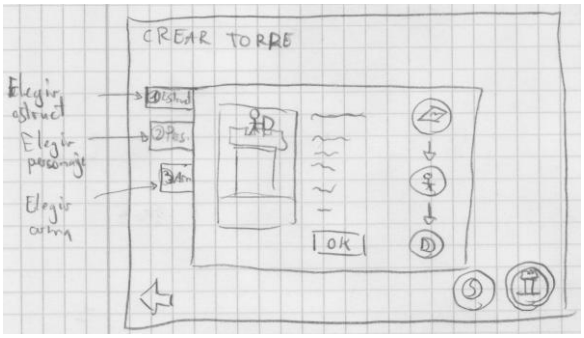
**Detalle de interfaz de usuario #6 – Compras de material y ofertas**



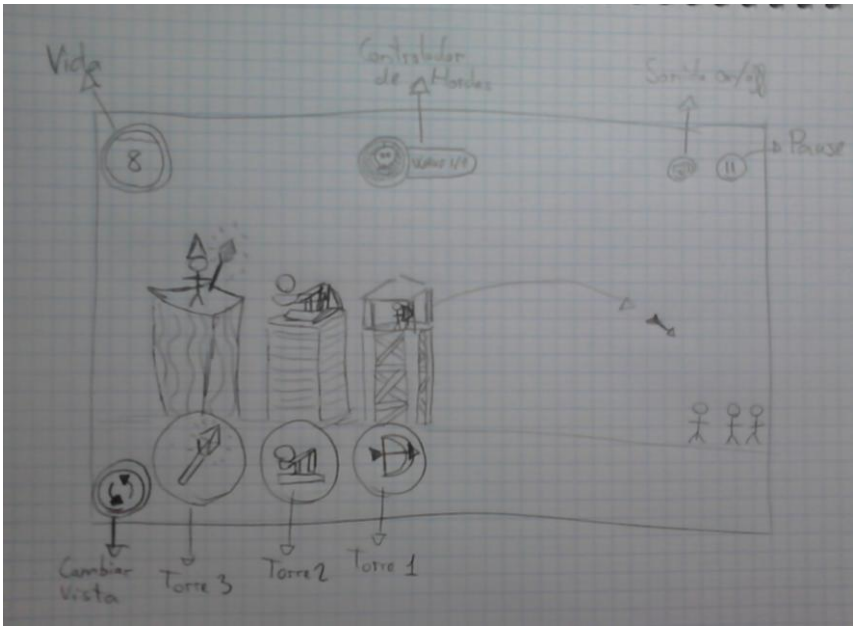

**Detalle de interfaz de usuario #7 – Editor de armas**




**Detalle de interfaz de usuario #8 – Creación y mezcla de torres por pasos**



## Detalle de interfaz de usuario #9 – Interfaz Gameplay




## Detalle de interfaz de usuario #10 – Interfaz torre

# Armas

## Visión general

Nuestras armas serán las diferentes torres que manejemos. Y podrán ser de 3 tipos: especial, pesadas y ligeras.

Las torres especiales tendrán el mayor tiempo de carga entre disparos de las tres. Serán las torres más grandes y utilizarán proyectiles especiales con diferentes efectos. Estás serán las torres que más diferencias tendrán entre ellas mismas. Sus trayectorias serán muy dependientes de la torre y podrán ser desde un barrido por toda la pantalla a una zona concreta. Por ejemplo en esta categoría podrían entrar dragones o magos que lancen como proyectil fuego o hechizos focalizados.

Las torres pesadas tendrán un tiempo de recarga menos que las especiales y serán un poco más pequeñas. Los proyectiles utilizados por estas torres tendrán un mayor peso y mucha más resistencia al viento con lo que tendrán trayectorias físicas más robustas respecto al viento. Por ejemplo en esta categoría entrarían catapultas, trabucos o cañones.

Las torres ligeras tendrán el tiempo de recarga más corto y serán más pequeñas que las anteriores. Los proyectiles utilizados por estas torres tendrán menor peso y serán muy susceptibles al viento. Por ejemplo en esta categoría entrarían arqueros o lanceros

## Detalles de armas #1

Cada torre se divide en 3 partes:

-Estructura: La estructura definirá el tipo de torre, siguiendo los 3 tipos explicados anteriormente.

-Personaje: Según el tipo de torre, podremos elegir un personaje u otro, que se encargarán de disparar los proyectiles.

-Arma: Por medio del arma se definirá el tipo de trayectoria, proyectil, etc.

Cada parte podrá combinarse con los diferentes elementos para crear muchos y variados tipos de torres únicos, añadiendo distintos porcentajes y tipos de daños.

Como ejemplo se describirán algunas posibles combinaciones

| Planos | Elemento | Resultado |
|---|---|---|
| Grande | Madera | Torreón de madera |
| Mediano | Metal | Torre de metal |
| Pequeño | Madera | Torreta de madera |

| Raza | Elemento | Resultado |
|---|---|---|
| Humano | Energia | Mago |
| Humano | Tierra | Enano |
| Enano | Metal | Golem |

| Armas | Elemento | Resultado |
|---|---|---|
| Arco | Fuego | Arco de fuego |
| Catapulta | Hielo | Catapulta de hielo |
| Trabuco | Veneno | Trabuco envenenado |

# Partituras musicales y efectos de sonido

## Visión general

La banda sonora del juego y los efectos de sonido serán sencillos, ya que en este juego queremos hacer un especial enfoque a mejorar la jugabilidad con respecto a otros juegos del mismo tipo. Cada parte del juego(ejemplo, UI de selección de modos de juego, niveles, etc…) tendrá una banda sonora diferente, así como cada acción dentro del modo de juego tendrá un efecto de sonido que haga que el usuario pueda meterse de pleno en el gameplay.

## Diseño del sonido

Salvo necesidad, se utilizarán principalmente bancos de sonidos ya creados y bandas sonoras compuestas y disponibles para su uso. Se utilizará las bandas sonoras y música de fondo de esta manera:

-Una específica para la pantalla de inicio y selección de modos de juego.

-Cada nivel tendrá su propia música de fondo, asociada a la batalla. Si la duración es muy larga podrían incluirse hasta dos piezas musicales diferentes en el mismo nivel.

-El editor de armas también tendrá una pista musical específica. Al poderse editar armas tras obtener las puntuaciones, compartirá música de fondo con la pantalla de puntuaciones tras vencer un nivel.

-Las introducciones históricas también tendrán su propio fondo musical.

Con respecto a los efectos de sonido:

-Los monstruos y enemigos irán agrupados por tipos, y cada tipo tendrá su propio efecto sonoro cuando sea golpeado.

-Al inicio de cada ronda, efectos sonoros propios de la llegada de un ejército y el comienzo de una batalla tendrán lugar.

-Durante una batalla, se reproducirán sonidos de fondo comunes en una batalla(gritos, choques de espadas, etc…)

-Cada disparo de un arma del usuario tendrá también su propio sonido, agrupados como los enemigos.

-Al vencer un nivel, se reproducirá un efecto sonoro correspondiente.

Con respecto a la interfaz:

-Todos los botones  tendrán un sonido leve que sonara cada vez que se pase por este. De esta forma sabremos que toda interfaz que haga sonido al pasar será un elemento interactivo.

- Los elementos del editor tendrán sonidos para señalar que la mezcla ha salido bien o mal. Estos sonidos serán distintos e identificativos.

-Cada vez que se seleccione un elemento a mezclar se generará un sonido distinto para dejar claro que se ha seleccionado un elemento.

# Juego para un jugador

## Visión general

El juego está totalmente enfocado al disfrute de un solo jugador, ya que es el principal potencial de este tipo de juegos. La experiencia del usuario se basará en jugar por niveles, donde cada nivel tendrá un escenario en el cual habrá un elemento a defender,  diferente en cada uno.

Cada nivel tendrá diferentes rondas. Después de cada ronda el usuario podrá, mediante la puntuación conseguida, retocar sus defensas para prepararse contra la siguiente oleada de enemigos que aparecerá en la ronda próxima hasta que llegue la última ronda, si la hay.

Así pues, el usuario tendrá que situar  los diferentes elementos de defensa antes de cada ronda de entre las diferentes opciones que tiene disponibles para ese escenario, y colocarlos en función de las posiciones para cada tipo de elemento y como quiera defender su territorio.

## Detalles del juego para un jugador #1

El principal modo de juego será el de campaña. Tras seleccionar la opción de jugar, los diferentes escenarios irán sucediéndose como una historia. Conforme avance en la campaña se irán desbloqueando los diferentes escenarios de los que dispone el juego, pero será necesario que el jugador los supere completamente.

Se le proporcionará al usuario un conjunto de armas básicas de inicio con las cuales irá interactuando para superar los diferentes niveles, consiguiendo nuevas a lo largo del modo de juego de diferentes formas.

## Detalles del juego para un jugador #2

Tras haber jugado la campaña y desbloqueado los escenarios, el jugador podrá elegir jugarlos por separado en un modo survival. Este modo consistirá en ir superando rondas sin un límite, en la que cada ronda será más difícil que el anterior hasta que sea derrotado.  De este modo podrá jugar hasta la saciedad su escenario favorito sin necesidad de pasar antes por el resto de escenarios anteriores.

## Detalles del juego para un jugador #3

En los dos modos anteriores habrá otra característica en común, el editor de armas. El jugador puede conseguir armas completas mediante compra o bien mediante la combinación de elementos separados que también podrán conseguirse y comprarse. De este modo puede disfrutar de una gran combinación de armas diferentes, utilizando las más convenientes en cada escenario. Dichas armas y sus combinaciones están explicadas en el apartado de armas de este mismo documento.

## Historia

El jugador se pondrá en la piel del mejor estratega de la Alianza de los Doce, una unión de doce reinos humanos, elfos y enanos que antaño se unieron para ganar la Guerra de los 100 Siglos que acabó con los conflictos entre la alianza y el ejército de las bestias provenientes de una isla más allá del mar que los humanos llaman Fin de La Costa , trayendo prosperidad y paz a los reinos que conformaban la alianza. Sin embargo con el tiempo, tras no obtener beneficios por la guerra, la corrupción y la codicia de algunos líderes empeoró la economía y las relaciones entre la unión.

En el momento más crítico, a punto de la disolución de la Alianza de los Doce y una posible guerra civil, un nuevo líder aparece entre las bestias y lidera un ataque a la Península con el fin de destruir todo a su paso y sumir en la desesperación a todas la razas. Debido al miedo y a la influencia del nuevo líder, algunos reinos de las diferentes razas caen rendidos a su poder y traicionan a la alianza, enfrentándose a aquellos que no han sucumbido a las tinieblas y que hacen todo lo posible por resistir. Cuando todo parece perdido, en el ataque a un pequeño pueblo con talento para la creación de armas, el jugador liderará la resistencia, comenzando con una campaña de reconquista que le convertirá en el mejor general de la historia, consiguiendo de nuevo el apoyo de todos los reinos y devolviendo de nuevo la esperanza y la prosperidad a la Alianza de los Doce.

## Horas de juego

El modo campaña durará aproximadamente 3 horas, teniendo en cuenta el número de escenarios y la duración de cada ronda y las oleadas de enemigos de cada una. Puede alargarse o acortarse según el dominio del usuario, su conocimiento acerca de las armas y el tiempo que dedique a obtener nuevas combinaciones.

Sin embargo puede dedicarle muchas más horas de juego mediante el modo survival. En concreto, las que el usuario quiera.

## Condiciones de victoria

El jugador obtendrá la victoria cuando haya completado todos los niveles del modo de campaña. Una vez obtenida la victoria tendrá total acceso a los escenarios y las armas disponibles, pudiendo repetir el modo campaña o jugar cada nivel por separado en el modo survival.

# Renderizado de personajes

## Visión general

A continuación se mostrarán los modelos elegidos para representar las clases y enemigos descritos anteriormente.

### Renderizado: Torre de madera

**Renderizado: Humano**

**Renderizado: Goblin**

**Renderizado: Trol**

**Renderizado: Zombie**

**Renderizado: Demonio de hielo**

**Renderizado: Demonio de fuego**

**Renderizado: Demonio oscuro**

**Renderizado:  Demonio de tierra**

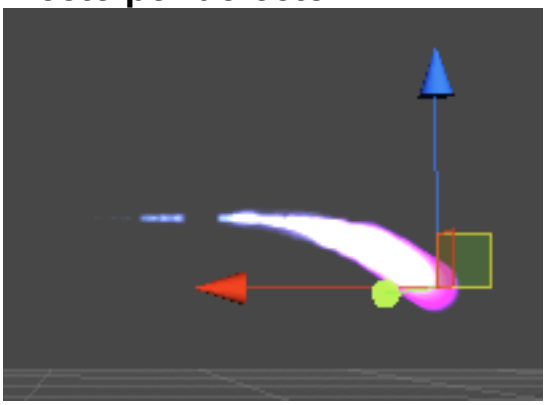
**Renderizado: Golem de hielo**

# Efectos elementales

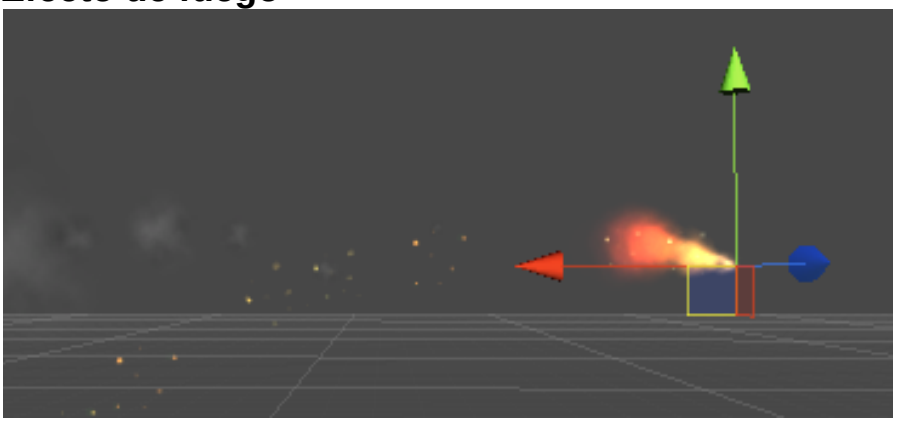
## Visión general

Aquí se podrán ver los diferentes efectos que tiene un arma al ser embebida con un elemento.

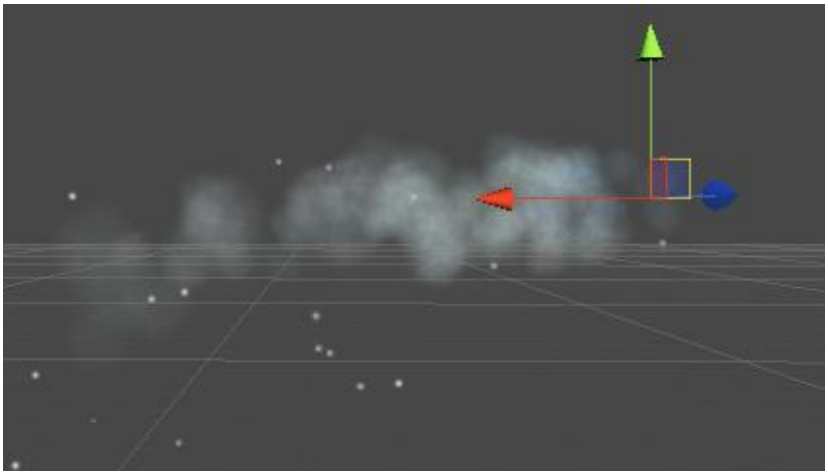
### Efecto por defecto



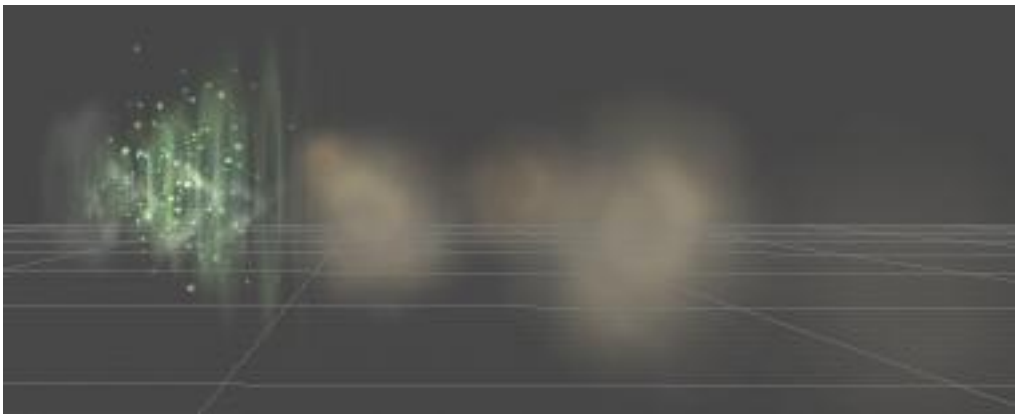
### Efecto de fuego



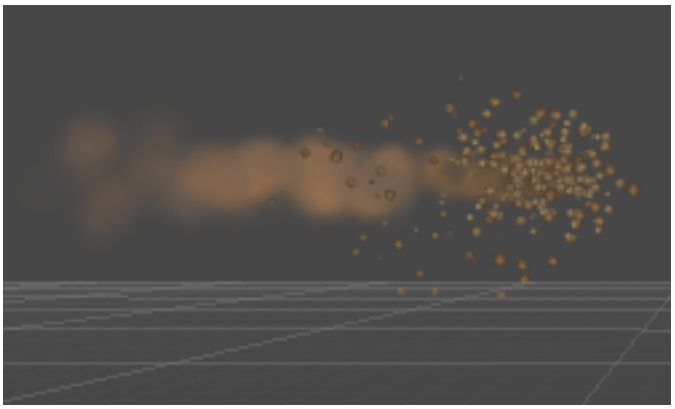
**Efecto de hielo**



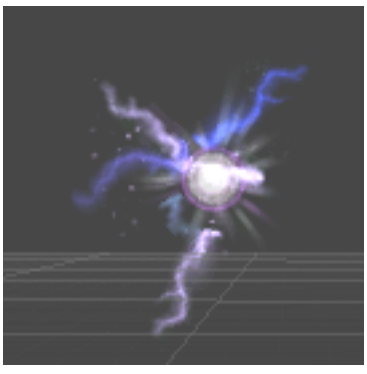
**Efecto de viento**



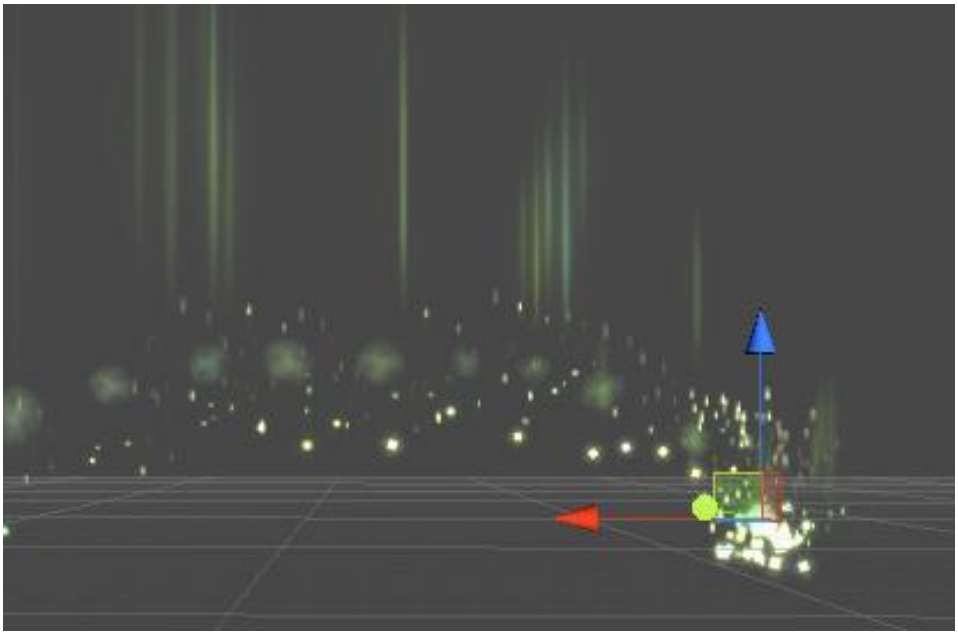
**Efecto de tierra**



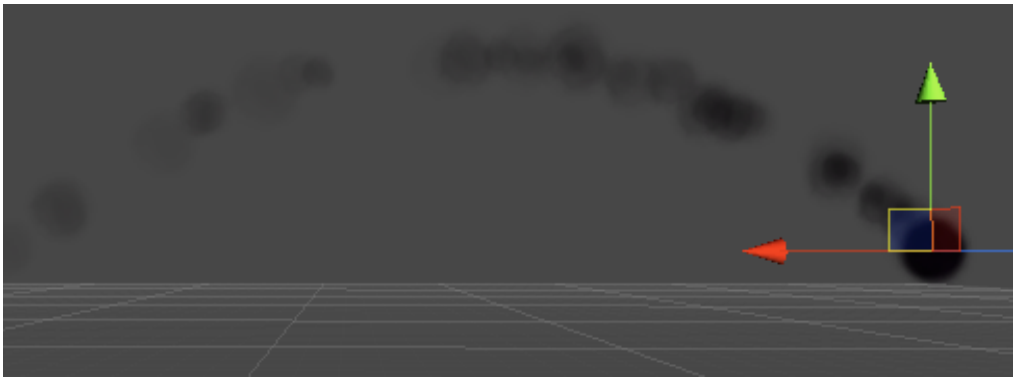
**Efecto de electricidad**



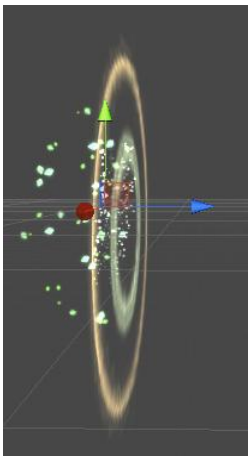
### Efecto de luz

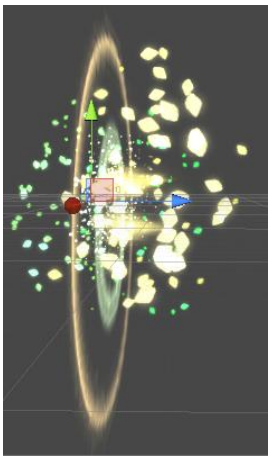


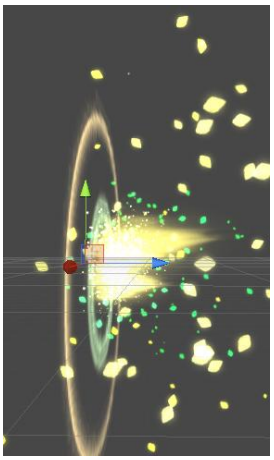
### Efecto de oscuridad



### Efecto de barrera







# “Apéndice de combinación de elementos”

Humano + Oscuridad = Goblin

Humano + Viento = Elfo

Humano + Tierra = Enano

Humano + Luz = Ángel

Humano + Electricidad = Mago humano

Elfo + Viento = Mago Elfo

Enano + Tierra = Mago enano

Arco + Fuego = Arco de fuego

Arco + Hielo = Arco de hielo

Arco + Viento = Arco de viento

Arco + Tierra = Arco de tierra

Arco + Electricidad = Arco de electricidad

Arco + Luz = Arco de luz

Arco + Oscuridad = Arco de oscuridad

Lanza + Fuego = Lanza de fuego

Lanza + Hielo = Lanza de hielo

Lanza + Viento = Lanza de viento

Lanza + Tierra = Lanza de tierra

Lanza + Electricidad = Lanza de electricidad

Lanza + Luz = Lanza de luz

Lanza + Oscuridad = Lanza de oscuridad

Plano pequeño + Madera = Torre pequeña de madera

Plano pequeño + Metal = Torre pequeña de metal

Plano pequeño + Piedra = Torre pequeña de Piedra

Plano mediano + Madera = Torre mediana de madera

Plano mediano + Metal = Torre mediana de metal

Plano mediano + Piedra = Torre mediana de Piedra

Plano grande + Madera = Torre grande de madera

Plano grande + Metal = Torre grande de metal

Plano grande + Piedra = Torre grande de Piedra